PETSc-FEM: A General Purpose, Parallel, Multi-Physics FEM Program. User's Guide

by Mario Storti, Norberto Nigro, Rodrigo Paz, Lisandro Dalcín and Ezequiel López

Centro Internacional de Métodos Computacionales en Ingeniería (CIMEC) Santa Fe, Argentina

http://www.cimec.org.ar/petscfem

((version "mstorti-v15-root-5-g62b6839 'clean")

(date "Sun Apr 27 18:22:33 2008 -0300")

(processed-date "Sun Apr 27 18:22:36 2008 -0300"))

April 27, 2008

This is the documentation for PETSc-FEM (current version `mstorti-v15-root-5-g62b6839 'clean`, a general purpose, parallel, multi-physics FEM program for CFD applications based on PETSc. PETSc-FEM comprises both a library that allows the user to develop FEM (or FEM-like, i.e. non-structured mesh oriented) programs, and a suite of application programs. It is written in the C++ language with an OOP (Object Oriented Programming) philosophy, but always keeping in mind the scope of efficiency.

Contents

1 LICENSE **6**

2 GNU GENERAL PUBLIC LICENSE **6**
- 2.1 Preamble . . . 6
- 2.2 GNU General Public License. Terms and Conditions for Copying, Distribution and Modification . . . 7
- 2.3 Appendix: How to Apply These Terms to Your New Programs . . . 11

3 The PETSc-FEM philosophy **13**
- 3.1 The three levels of interaction with PETSc-FEM . . . 13
- 3.2 The elemset concept . . . 13

4 General layout of the user input data file **14**
- 4.1 Preprocessing the user input data file . . . 15
- 4.2 Internal preprocessing. The FileStack class . . . 16
  - 4.2.1 Syntax . . . 16
  - 4.2.2 Class internals . . . 17
- 4.3 Preprocessing with ePerl . . . 17
  - 4.3.1 Basics of ePerl . . . 17
  - 4.3.2 Variables . . . 18
  - 4.3.3 Text expansion . . . 18
  - 4.3.4 Conditional processing . . . 20
  - 4.3.5 File inclusion . . . 20
  - 4.3.6 Use of ePerl in makefiles . . . 21
  - 4.3.7 ePerlini library . . . 21
  - 4.3.8 Errors in ePerl processing . . . 22
- 4.4 General options . . . 22
  - 4.4.1 Read mesh options . . . 22
  - 4.4.2 Elemset options . . . 23
  - 4.4.3 PFMat/IISDMat class options . . . 24
- 4.5 Emacs tools and tips for editing data files . . . 26
  - 4.5.1 Installing PETSc-FEM mode . . . 27
  - 4.5.2 Copying and pasting PETSc-FEM options . . . 27

5 Text hash tables **28**
- 5.1 The elemset hash table of properties . . . 28
- 5.2 Text hash table inclusion . . . 29
- 5.3 The global table . . . 30
- 5.4 Reading strings directly from the hash table . . . 31
- 5.5 Reading with 'get_int' and 'get_double' . . . 31
- 5.6 Per element properties table . . . 31
- 5.7 Taking values transparently from hash table or per element table . . . 32

6 The general advective elemset **34**
6.1 Introduction to advective systems of equations . . . 34
6.2 Discretization of advective systems . . . 35
6.3 SUPG stabilization . . . 36
6.4 Shock capturing . . . 36
6.5 Creating a new advective system . . . 37
6.6 Flux function routine arguments . . . 37
6.6.1 Options . . . 39
6.7 The hydrology module . . . 42
6.7.1 Related Options . . . 44
6.8 The Hydrological Model (cont.). . . . 45
6.9 Subsurface Flow. . . . 46
6.10 Surface Flow. . . . 46
6.10.1 2D Saint-Venant Model. . . . 46
6.10.2 1D Saint-Venant Model. . . . 47
6.10.3 Kinematic Wave Model. . . . 47
6.11 Boundary Conditions. . . . 48
6.11.1 Boundary Conditions to simulate River-Aquifer Interactions/Coupling Term. . . . 48
6.11.2 Initial Conditions. First, Second and Third Kind Boundary Conditions/Absorbent Boundary Condition. . . . 48
6.12 Absorbing boundary conditions . . . 50
6.12.1 Linear absorbing boundary conditions . . . 51
6.12.2 Riemann based absorbing boundary conditions . . . 52
6.12.3 Absorbing boundary conditions based on last state . . . 53
6.12.4 Finite element setup . . . 53
6.12.5 Extrapolation from interiors . . . 54
6.12.6 Avoiding extrapolation . . . 55
6.12.7 Flux functions with enthalpy. . . . 56
6.12.8 Absorbing boundary conditions available . . . 57

7 The Navier-Stokes module **59**
7.1 LES implementation . . . 59
7.1.1 The wall elemset . . . 60
7.1.2 The mixed type boundary condition . . . 60
7.1.3 The van Driest damping factor. Programming notes . . . 61
7.2 Options . . . 62
7.3 Mesh movement . . . 69

8 Tests and examples **71**
8.1 Flow in the anular region between to cylinders . . . 72
8.2 Flow in a square with periodic boundary conditions . . . 72
8.3 The oscilating plate problem . . . 72
8.4 Linear advection-diffusion in a rectangle . . . 74

9 The FastMat2 matrix class **74**
9.1 Introduction . . . 74
9.2 Example . . . 75
9.2.1 Current Matrix view . . . 76
9.2.2 Set operations . . . 77
9.2.3 Dimension matching . . . 77
9.2.4 Automatic dimensioning . . . 77
9.2.5 Concatenation of operations . . . 77
9.3 Caching the adresses used in the operations . . . 77
9.3.1 Branching . . . 78
9.3.2 Loops executed a non constant number of times . . . 82
9.3.3 Masks can't traverse branches . . . 84
9.3.4 Efficiency . . . 84
9.4 Synopsis of operations . . . 85
9.4.1 One-to-one operations . . . 85
9.4.2 In-place operations . . . 85
9.4.3 Generic "sum" operations (sum over indices) . . . 86
9.4.4 Sum operations over all indices . . . 87
9.4.5 Export/Import operations . . . 87
9.4.6 Static cache operations . . . 88

10 Hooks **88**
10.1 Launching hooks. The hook list . . . 89
10.2 Dynamically loaded hooks . . . 89
10.3 Shell hook . . . 90
10.4 Shell hooks with "make" . . . 92

11 Gatherers and embedded gatherers **93**
11.1 Dimensioning the values vector . . . 94
11.2 Embedded gatherers . . . 94
11.3 Automatic computation of layer connectivities . . . 97
11.4 Passing element contributions as per-element properties . . . 98
11.5 Parallel aspects . . . 99
11.6 Creating a gatherer . . . 99

12 Generic load elemsets **100**
12.1 Linear generic load elemset . . . 100
12.2 Functional extensions of the elemset . . . 101
12.3 The flow reversal elemset . . . 101

13 Visualization with DX **102**
13.1 Asynchronous/synchronous communication . . . 104
13.2 Building and loading the ExtProgImport module . . . 104
13.3 Inputs/outputs of the ExtProgImport module . . . 105
13.4 DX hook options . . . 105

14 The "idmap" class **106**
14.1 Permutation matrices . . . 106
14.2 Permutation matrices in the FEM context . . . 107
14.3 A small example . . . 111
14.4 Inversion of the map . . . 114
14.5 Design and efficiency restrictions . . . 114
14.6 Implementation . . . 114
14.7 Block matrices . . . 115
14.7.1 Example: . . . 115
14.8 Temporal dependent boundary conditions . . . 116
14.8.1 Built in temporal functions . . . 117
14.8.2 Implementation details . . . 122
14.8.3 How to add a new temporal function . . . 122
14.8.4 Dynamically loaded amplitude functions . . . 123
14.8.5 Use of prefixes . . . 125
14.8.6 Time like problems. . . . 127

15 The compute_prof package **127**
15.1 MPI matrices in PETSc . . . 127
15.2 Profile determination . . . 128

16 The PFMat class **129**
16.1 The PFMat abstract interface . . . 129
16.2 IISD solver . . . 129
16.2.1 Interface preconditioning . . . 131
16.3 Implementation details of the IISD solver . . . 132
16.4 Efficiency considerations . . . 134

17 The DistMap class **135**
17.1 Abstract interface . . . 136
17.2 Implementation details . . . 136
17.3 Mesh refinement . . . 138
17.3.1 Symmetry group generator . . . 139
17.3.2 Canonical ordering . . . 141
17.4 Permutation tree . . . 141
17.5 Canonical ordering . . . 142
17.6 Object hashing . . . 142

18 Synchronized buffer **144**
18.1 A more specialized class . . . 146

19 Authors **147**

20 Grants received **148**

21 Symbols and Acronyms **150**
21.1 Acronyms . . . 150

1 LICENSE

The PETSc - FEM package is a library and application suite oriented to the Finite Element Method based on PETSc. Copyright (C) 1999-2008, Mario Alberto Storti, Norberto Marcelo Nigro, Rodrigo R. Paz, Lisandro Dalcin, Ezequiel Lopez. Centro Internacional de Metodos Numericos en Ingenieria (CIMEC-Argentina), Universidad Nacional del Litoral (UNL-Argentina), Consejo Nacional de Investigaciones Cientificas y Tecnicas (UNL-Argentina).

This program is free software; you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program; if not, write to the Free Software Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307, USA.

2 GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Copyright (C) 1989, 1991 Free Software Foundation, Inc. 675 Mass Ave, Cambridge, MA 02139, USA Everyone is permitted to copy and distribute verbatim copies of this license document, but changing it is not allowed.

2.1 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too,

receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps:

- copyright the software, and
- offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

2.2 GNU General Public License. Terms and Conditions for Copying, Distribution and Modification

0. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

1. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or

work under the terms of Section 1 above, provided that you also meet all of these conditions:

a) You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

b) You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

c) If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

a) Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

b) Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

c) Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

4. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

5. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

6. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.

7. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

8. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

9. The Free Software Foundation may publish revised and/or new versions of the General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

10. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

11. WARRANTY BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND

PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

12. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

2.3 Appendix: How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
<one line to give the  program's name and a brief  idea of what it
does.> Copyright (C) 19yy <name of author>

This program  is  free software;  you can  redistribute  it and/or
modify it under  the terms  of  the GNU General  Public License as
published by the Free Software Foundation; either version 2 of the
License, or (at your option) any later version.

This  program is distributed in the  hope that  it will be useful,
but WITHOUT   ANY WARRANTY; without  even the  implied warranty of
MERCHANTABILITY or FITNESS FOR  A PARTICULAR PURPOSE.  See the GNU
General Public License for more details.

You should have received a copy of the  GNU General Public License
along  with  this program;  if  not, write   to  the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details type
‘show w’. This is free software, and you are welcome to
redistribute it under certain conditions; type ‘show c’ for
details.
```

The hypothetical commands ‘show w’ and ‘show c’ should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than ‘show w’ and ‘show c’; they could even be mouse-clicks or menu items–whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a ”copyright disclaimer” for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in the
program ‘Gnomovision’ (which makes passes at compilers) written by
James Hacker.

<signature of Ty Coon>, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

3 The PETSc-FEM philosophy

3.1 The three levels of interaction with PETSc-FEM

As stated in the PETSc-FEM description, it is both a library and an application suite. That means that some applications as Navier-Stokes, Euler (inviscid flow), shallow water, general advective linear systems and the Laplace/Poisson equation, come bundled with it, whereas the library is an abstract interface that allows people to write other applications. So that we distinguish between the *"user"* for which the interaction with PETSc-FEM is limited to writing data files for the bundled applications, from the *"application writers"* that is people that uses the library to develop new applications. Usually, application writers write a `main()` routine that use routine calls to the PETSc-FEM library in order to assemble PETSc vectors and matrices and perform algebraic operations among them via calls to PETSc routines. In addition, they also have to code *"element routines"* that compute vectors and matrices at the element level. PETSc-FEM is the code layer that is in charge of assembling the individual contributions in the global vectors or matrices, taking into account fixations, etc... Finally, there is the *"PETSc-FEM programmers"*, that is people that write code for the core library.

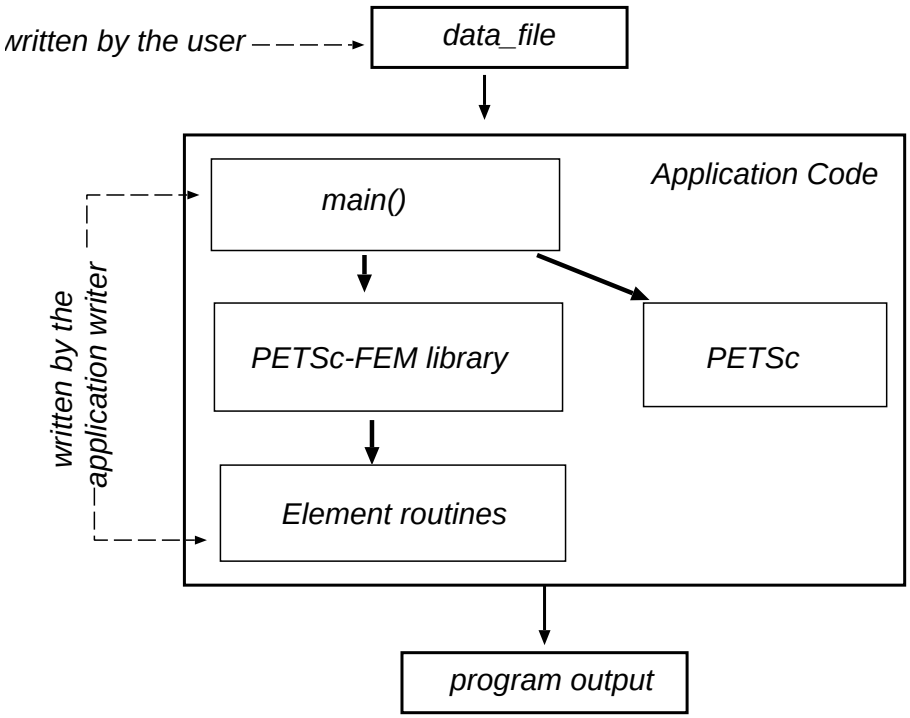


Figure 1: Typical structure of a PETSc-FEM application

3.2 The elemset concept

PETSc-FEM is written in the C++ language and sticks to the Object-Oriented Programming (OOP) philosophy. In OOP, data is stored in *"objects"* and the programmer access them via an abstract interface. A first approach to writing a Finite Element program with

OOP philosophy, is to define each element or node as an object. However, this is both time and memory consuming, since accessing each element or each node is performed by passing through the whole code layer of the element or node class. As one of the primary objectives of PETSc-FEM is the efficiency, we solved this by defining the basic objects as a whole set of elements of nodes that share almost all the properties, aside element connectivities or node coordinates. This is very common in CFD, where for each problem almost all the elements share the same physical properties (viscosity, density, etc...) and options (number of integration Gauss points, parameters for the FEM formulation, etc...). Thus, for each problem the user defines a `nodedata` object, and one or several `elemset` objects.

Each elemset groups elements of the same type, i.e. for which residuals and matrices are to be computed by the same routine. Usually, in CFD all the elements are processed by the same routine, so that one may ask for what it may serve to have several elemsets. First, some boundary conditions (constant flux or mixed boundary conditions for heat conduction, absorbing boundary conditions for advective systems) may be imposed more conveniently through an elemset abstraction. Also, several elemsets may be used for reducing the space required for storing varying physical properties. If some physical property is shared by all the elements, for instance viscosity or specific heat, then it is defined once for all the elemset. If the quantity varies continuously in the region, but it is known a priori, then it can be passed as a "per-element" property (see §5.6), but that means storage of a double (or the amount of memory needed for that property) for each element. If it is not the same on all the mesh, but is constant over large regions, then it may be convenient to divide the mesh in several elemsets, where the given property has the same value over the elements of each elemset.

4 General layout of the user input data file

Input to application packages (like `ns`, `advective` or `laplace`) is feed via input data files, usually with extension `.dat`. This file contains global options, node coordinates, element connectivities, physical properties, fixations and boundary conditions, etc... Even if the precise format of the file may be changing it is worth to describe the general layout.

The file is divided in sections: the `table` sections, the `nodedata` section, several `elemset` sections, the `fixa` section and `constraint` section.

Each section starts with the keyword identifying the section in a separate line, followed by several parameters in the same line. Follows several lines that makes the section, ending with a terminator of the form `__END_<`*some-identifier*`>__`, for instance `__END_HASH__`. (Note that these terminators start and end with double underscores (`__`) while single underscores are used to separate words inside the keyword). For instance, an elemset section is of the form

```
elemset volume_euler 4

geometry cartesian2d
ndim 2
npg 4
chunk_size 1000
```

```
lumped_mass 1
shock_capturing 1
gamma 1.4
<... other element options go here >
__END_HASH__
    1    2   81   80
    2    3   82   81
    3    4   83   82
    4    5   84   83
    5    6   85   84
    6    7   86   85
    7    8   87   86
    8    9   88   87
<... more element connectivities follow here >
__END_ELEMSET__
```

In this example, the keyword `elemset` is followed by the parameters `volume_euler` that is the elemset type, and `4` that is the number of nodes connected to each element. The line starting with `props` describes some per-element quantities (more on this later, see §5.4). Follows the assignation of some values to parameters for the actual elemset, for instance the value `4` is assigned to the `npg` parameter, that is, the number of Gauss points. The assignation of parameters ends with the `__END_HASH__` terminator. Follows the element connectivities, one per line, ending with the terminator `__END_ELEMSET__`.

4.1 Preprocessing the user input data file

It's very handy to have some preprocessing capabilitites when reading input files, for instance including files, if-else constructs, macro definitions, inline arithmetics, etc... Some degree of preprocessing is performed inside PETSc-FEM – this includes file inclusion, skipping comments, and continuation lines and is described in section §4.2. Off course, this internal preprocessing may be combined with any previous preprocessing package, such as `m4` or `ePerl`. In section §4.2 we describe internal preprocessing while in section§4.3 we describe preprocessing with ePerl.

The reason to have this two mechanisms of preprocessing is the following. Preprocessing with ePerl (or m4, or any other packages) is very powerful and well supported, however the mechanism is to create an intermediate file, and this file may be very large for large problems, so that some internal preprocessing including, at least, the file inclusion is needed. On the other hand, including all the preprocessing capabilities of preprocessing as in ePerl is beyond the scope of PETSc-FEM, so that we preferred to keep with this two levels of preprocessing. The idea is to have a small user data file where the strong capabilities of an external preprocessing package may be used while performing file inclusion of very large files containing node coordinates, element connectivities and so on, as the file is reading, avoiding the creation of large intermediate files. In addition, this allows the user to choose the external preprocessing package.

4.2 Internal preprocessing. The FileStack class

This class allows reading from a set of linked files (the PETSc-FEM data file including node coordinates, mesh connectivities, etc...) with some preprocessing capabilities. Supports file inclusion, comments and continuation lines. The preprocessing capabilities supported in this class are the minimal ones needed in order to treat very large files efficiently. More advanced preprocessing capabilities will be added in a higher level layer written in Perl or similar (may be ePerl?).

4.2.1 Syntax

The syntax of comments and continuation lines is borrowed from Unix shells,

Comments: From the first appearance of "`#`" to the end of the lines is considered a comment.

Continuation lines: If a line ends in "\", (i.e. if backslash is the last character in the line, before newline "`^J`") then the next line is appended to the previous one to form a logical single line.

File inclusion: The directive

```
__INCLUDE__ path/to/file
```

inserts the contents of "`file`" in directory "`path/to/`" to this file in the actual point. Directories may be absolute or relative (we use "`fopen`" from "`stdio`"). File inclusion may be recursive to the limit of the quantity of files that can be kept open simultaneously by the system.

Echo control: The directives `__ECHO_ON__`, `__ECHO_OFF__` controls whether the lines read from input should be copied to the output. Usually one may be interested in copying some part of the input to the output in order to remember the parameters of the run. As implemented so far, this feature is recursive so that if included files (with the internal preprocessing, i.e. with the `__INCLUDE__` directive) will be also copied to the output, unless you enclose the `__INCLUDE__` line itself with a `__ECHO_OFF__`, `__ECHO_ON__` pair. For instance

```
__ECHO_ON__
global_options
...             # more options here
nsaverot 50
viscosity 13.3333333333333
weak_form 0
...             # and here
__END_HASH__

nodes  2  2  3
                # do not echo coordinates
```

```
__ECHO_OFF__
__INCLUDE__ stabi.nod.tmp
__ECHO_ON__
__END_NODES__
...
```

4.2.2 Class internals

As its names suggests, the class is based in a stack of files. When a "`get_line()`" is issued a new line is read, comments are skipped and continuation lines are resolved. if the line is an "`__INCLUDE__`" directive, then the current line is "pushed" in the stack and the new file is open and is the current file.

4.3 Preprocessing with ePerl

ePerl (for *"embedded Perl"*) is a package that allows inclusion of Perl commands within text files. Perl is the *"Practical Extraction and Report Language"*, a scripting language optimized for scanning arbitrary text files, extracting information from those text files, and printing reports based on that information. For more information about ePerl see http://www.engelschall.com/sw/eperl/, while for more information on Perl see http://www.perl.com.

Describing the possibilities of preprocessing with ePerl are far beyond the scope of this manual. We will describe some basic techniques of use in the context of writing PETSc-FEM user data files.

4.3.1 Basics of ePerl

ePerl allow you to embed Perl commands within text, enclosing them within `<:` *and* `:>` delimiters, for instance

```
<: $pi = 2*atan2(1,0); $theta = sin($pi/4); :>
...
some text here
...

theta <: print $theta :>
beta <: print 4*$theta :>
```

results, after being processed by ePerl in

```
...
some text here
...

theta 0.707106781186547
beta 2.82842712474619
```

The basic rules are

- Variables start with `$` following by a C-like identifier (alphanumeric plus underscore, case sensitive), for instance `$alpha` or `$my_variable`.
- Statements end in semicolon.
- The text inserted in place of the ePerl block is the output of the commands inside the block. This output is done in Perl with the `print()` function, but in ePerl there is a shortcut of the form `<:=`*expression*`:>`.
- Mathematical functions `sin`, `cos`, `tan`, `exp`, `atan2(y,x)` are available, as in C, powers x^y are expressed as `x**y` (i.e. Fortran like).

4.3.2 Variables

You can define variables to use them after in different places, and also in mathematical expressions

```
<: $Reynolds = 1000; $L = 2; $rho = 1.345;
   $velocity=3.54; $Grashof=6.e4; :>
...
mu <:=$rho*$velocity*$L/$Reynolds:>
Nusselt <:=(($Reynolds*$Grashof)**0.25):>
```

which results in

```
...
mu 0.0095226
Nusselt 88.0111736793393
```

4.3.3 Text expansion

It is common to have several lines of text that have to be repeated several times. For instance some options that hae to be applied to several elemsets. The trick is to assign the text to a variable via the "here-in document" `<<EOT` feature and then inserting in the appropriate places, for instance

```
<: $common_options = <<EOT;
option1   value1
option2   value 2
EOT
_:>

...

elemset type1 4
props
```

```
<:=$common_options:>
option3 value3

...
__END_ELEMSET__

elemset type2 3
props
<:=$common_options:>
option4 value4

...
__END_ELEMSET__
```

that expands to

```
...

elemset type1 4
props
option1   value1
option2   value 2

option3 value3

...
__END_ELEMSET__

elemset type2 3
props
option1   value1
option2   value 2

option4 value4

...
__END_ELEMSET__
```

Note the use of the underscore just before the `:>` terminator in the first ePerl block. This tells to ePerl not to include the semicolon terminator (see the ePerl documentation for further details.) The terminator `EOT` stands for "End Of Text" and may be replaced by any similar string. It must appear in a line by itself at the end of the text to be included.

4.3.4 Conditional processing

ePerl allows conditional processing, as with the C preprocessor, with `#if-#else-#endif` constructs as in

```
<:$method = "iterative"; :>
...

#if $method eq "iterative"
maxits 100
tolerance 1e-2
#else
maxits 1
tolerance 1e-10
#endif
...
```

expands to

```
...

maxits 100
tolerance 1e-2
...
```

Also, lines starting with `#c` are dicarded as comments. Conditional preprocessing and comments are enabled with the "`-P`" flag, so that make sure you have this flag enabled when preprocessing the `.epl` file (probably in the `Makefile` file). Note that PETSc-FEM comments (those starting with numeral "`#`") may collide with the ePerl preprocessing directives, so that when commenting out lines in PETSc-FEM input files it is safer to leave a space between the "`#`" character and the commented text

```
# commented text          (OK)
#commented tex            (but dangerous!)
```

4.3.5 File inclusion

In addition to the inclusion allowed in the internal preprocessor via the `__INCLUDE__` command, ePerl has his own inclusion directive, for instance

```
some text
...
#include /home/mstorti/PETSC/petscfem/doc/options.txt
...
another text
```

and provided file `options.txt contains`

```
# File options.txt
opion1 value1
option2 value2
```

then the previous block expands to

```
some text
...
# File options.txt
opion1 value1
option2 value2
...
another text
```

Including with the internal preprocessing directive `__INCLUDE__` has the advantage of not creating an intermediate file. On the other hand, including with the ePerl directive, allows recursive ePerl preprocessing and more versatility in defining the inclusion path (with the `@INC` list, see Perl documentation).

4.3.6 Use of ePerl in makefiles

Usually user data files have extension `.dat`. When preprocessing with ePerl the convention is to use `.epl` suffix for the file written by the user with ePerl commands, i.e. the files to be preprocessed, and suffix `.depl` for the preprocessed file. A line in the Makefile of the form

```
%.depl: %.epl
         eperl -P $< > $@
```

assures the translation when needed.

4.3.7 ePerlini library

Some useful constants and functions are found in the file `eperlini.pl`. This may be included in the user data file with the following line

```
<:require 'eperlini.pl':>// # Initializes ePerl
```

It includes a definition for `$PI` ($=\pi$), trigonometric and hyperbolic functions, and others.

A common mistake when using preprocessing packages like ePerl is to edit manually the preprocessed file `.depl`, instead to edit the `.epl` file. In order to avoid this we write protect the `.depl` file, for instance the section in the Makefile is replaced by

```
%.depl: %.epl
        if [ -e $@ ] ; then chmod +w $@ ; rm $@ ; fi
        eperl -P $< > $@
        chmod -w $@
```

In addition, inclusion of the `eperlini.pl` library inserts the following comment in the included file

```
# DON'T EDIT MANUALLY THIS FILE !!!
# This files automatically generated by ePerl from
# the corresponding '.epl' file.
```

4.3.8 Errors in ePerl processing

If the preprocessing stage with ePerl gives some error (on `STDERR`) preprocessing is stopped no ePerl output is given. For instance, if you include a directive like

```
<:=atanh(2.):>
```

the output looks like

```
ePerl:Error: Perl runtime error (interpreter rc=255)

---- Contents of STDERR channel: ---------
atanh: argument x must be |x| < 1.
------------------------------------------
```

In such a case you have to fix the ePerl commands prior to any further debugging of the PETSc-FEM run.

4.4 General options

The following options apply to all the modules.

4.4.1 Read mesh options

This options are read in the `read_mesh()` routine

- `int additional_iprops` (default=0):
  int additional properties (used by the element routine) (found in file: `readmesh.cpp`)

- `int additional_props` (default=0):
  Additional properties (used by the element routine) (found in file: `readmesh.cpp`)
- `int check_dofmap_id` (default=0):
  Checks that the `idmap` has been correctly generated. (found in file: `readmesh.cpp`)
- `int debug_element_partitioning` (default=0):
  Prints element partitioning. (found in file: `readmesh.cpp`)
- `int local_store` (default=0):
  Defines a "locker" for each element (found in file: `readmesh.cpp`)
- `int max_partgraph_vertices` (default=INF):
  Maximum number of vertices admissible while computing the partitioning graph. (found in file: `readmesh.cpp`)

- `string partitioning_method` (default=`metis`):

  Set partitioning method. May be set to `metis` , `hitchhiking` , `nearest_neighbor` or `random` . (found in file: `readmesh.cpp`)

- `int print_dofmap_id` (default=`0`):

  Prints the dofmap `idmap` object. (found in file: `readmesh.cpp`)

- `int print_hostnames` (default=`0`):

  Print hostnames for nodes participating in this run (found in file: `readmesh.cpp`)

- `int print_partitioning_statistics` (default=`0`):

  Print graph statistics (found in file: `readmesh.cpp`)

4.4.2 Elemset options

This options are used in the `Elemset` class

- `int chunk_size` (default=`ELEM_CHUNK_SIZE`):

  Chunk size for the elemset. (found in file: `elemset.cpp`)

- `int debug_compute_prof` (default=`0`):

  Debug the process of building the matrix profile. (found in file: `elemset.cpp`)

- `int element_weight` (default=`1`):

  Element weight for the processor (found in file: `elemset.cpp`)

- `double epsilon_fdj` (default=`EPSILON_FDJ`):

  The increment in the variables in order to compute the finite difference approximation to the Jacobian. Should be order epsilon=sqrt(precision)*(typical magnitude of the variable). Normally, precision=1e-15 so that epsilon=1e-7*(typical magnitude of the variable) (found in file: `elemset.cpp`)

- `int print_local_chunk_size` (default=`0`):

  Print the local chunk size used for each elemset in each processor for each chunk. (found in file: `elemset.cpp`)

- `int report_assembly_time` (default=`0`):

  Debug the process of building the matrix profile. (found in file: `elemset.cpp`)

- `int report_consumed_time` (default=`0`):

  Report consumed time for the elemset. Useful for building the table of weights per processor. (found in file: `elemset.cpp`)

- `int report_consumed_time_stat` (default=`0`):

  Print statistics about time spent in communication and residual evaluation (found in file: `elemset.cpp`)

4.4.3 PFMat/IISDMat class options

This options are used in the `PFMat` class

- `int iisd_subpart` (default=`1`):

  Number of subpartitions inside each processor. (found in file: `iisdcr.cpp`)

- `int iisd_subpart_auto` (default=`0`):

  Choose automatically the number of subdomains so as to have approximately this number of unknowns per subdomain. (found in file: `iisdcr.cpp`)

- `int iisdmat_print_statistics` (default=`0`):

  Print dof statistics, number of dofs local and interface in each processor. (found in file: `iisdcr.cpp`)

- `double interface_full_preco_fill` (default=`1.`):

  The ILU fill to be used for the `A_II` problem if the ILU preconditioning is used (found in file: `iisdcr.cpp`)

- `int interface_full_preco_maxits` (default=`5`):

  Number of iters in solving the preconditioning for the interface problem when using `use_interface_full_preco` . (found in file: `iisdcr.cpp`)

- `string interface_full_preco_pc` (default=`jacobi`):

  Defines the preconditioning to be used for the solution of the diagonal interface problem (not the Schur problem) (found in file: `iisdcr.cpp`)

- `double interface_full_preco_relax_factor` (default=`1.`):

  The problem on the interface is solved with Richardson method with few iterations (normally 5). Richardon iteration may not converge in some cases and then we can help convergence using a relaxation factor ¡1 (found in file: `iisdcr.cpp`)

- `string local_solver` (default=`PETSc`):

  Chooses the local solver (may be "PETSc" or "SuperLU") (found in file: `iisdcr.cpp`)

- `int max_partgraph_vertices_proc` (default=`INF`):

  The maximum number of vertices in the coarse mesh for sub-partitioning the dof graph in the IISD matrix. (found in file: `iisdcr.cpp`)

- `double pc_lu_fill` (default=`5.`):

  PETSc parameter related to the efficiency in growing the factored profile. (found in file: `iisdcr.cpp`)

- `int print_Schur_matrix` (default=`0`):

  Print the Schur matrix (don't try this for big problems). (found in file: `iisdcr.cpp`)

- `int print_interface_full_preco_conv` (default=0):

  Flags whether or not print the convergence when solving the preconditioning for the interface problem when using `use_interface_full_preco` . (found in file: `iisdcr.cpp`)

- `int use_interface_full_preco` (default=0):

  Chooses the preconditioning operator. (found in file: `iisdcr.cpp`)

- `int use_interface_full_preco_nlay` (default=1):

  Number of layers in the preconditioning band (should be `nlay>=1` .) (found in file: `iisdcr.cpp`)

- `int asm_lblocks` (default=1):

  Chooses the number of local blocks in ASM (found in file:)

- `int asm_overlap` (default=1):

  Chooses the overlap of blocks in ASM (found in file:)

- `string asm_sub_ksp_type` (default=`preonly`):

  Chooses the preconditioning for block problems in ASM method. (found in file:)

- `string asm_sub_preco_type` (default=`ilu`):

  Chooses the preconditioning for block problems in ASM method. (found in file:)

- `string asm_type` (default=`restrict`):

  Chooses the restriction/extension type in ASM (found in file:)

- `double atol` (default=`1e-6`):

  Absolute tolerance to solve the monolithic linear system (Newton linear subiteration). (found in file:)

- `int compact_profile_graph_chunk_size` (default=0):

  Size of chunk for the dynamic vector used in computing the mstrix profile. (found in file:)

- `double dtol` (default=`1e+3`):

  Divergence tolerance to solve the monolithic linear system (Newton linear subiteration). (found in file:)

- `string gmres_orthogonalization` (default=`modified_gram_schmidt`):

  Orthogonalization method used in conjunction with GMRES. May be `unmodified_gram_schmidt`, `modified_gram_schmidt` or `ir_orthog` (default). (Iterative refinement). See PETSc documentation. (found in file:)

- `int Krylov_dim` (default=50):

  Krylov space dimension in solving the monolithic linear system (Newton linear subiteration) by GMRES. (found in file:)

- `string KSP_method` (default=`gmres`):

  Defines the KSP method (found in file:)

- `int maxits` (default=`Krylov_dim`):

  Maximum iteration number in solving the monolithic linear system (Newton linear subiteration). (found in file:)

- `string preco_side` (default=`<ksp-dependent>`):

  Uses right or left preconditioning. Default is `right` for GMRES. (found in file:)

- `string preco_type` (default=`jacobi`):

  Chooses the preconditioning operator. (found in file:)

- `int print_fsm_transition_info` (default=`0`):

  Print Finite State Machine transitions for PFPETScMat matrices. 1: print inmediately, 2: gather events (non immediate printing). (found in file:)

- `int print_internal_loop_conv` (default=`0`):

  Prints convergence in the solution of the GMRES iteration. (found in file:)

- `double rtol` (default=`1e-3`):

  Relative tolerance to solve the monolithic linear system (Newton linear subiteration). (found in file:)

- `int use_compact_profile` (default=`LINK_GRAPH`):

  Choice representation of the profile graph. Possible values are: 0) Adjacency graph classes based on STL map+set, demands too much memory, CPU time OK. 1) Based on dynamic vector of pair of indices with resorting, demands too much CPU time, RAM is OK 2) For each vertex wee keep a linked list of cells containing the adjacent nodes. Each insertion is $O(m^2)$ where m is the average number of adjacent vertices. This seems to be optimal for FEM connectivities. (found in file:)

4.5 Emacs tools and tips for editing data files

GNU Emacs (http://www.gnu.org/software/emacs/) is a powerful text editor, that can colorize and indent text files according to the syntax of the language you are editing. Emacs has *"modes"* for most languages (C/C++, Pascal, Fortran, Lisp, Perl, ...). We have written a basic mode for PETSc-FEM data files that is distributed with PETSc-FEM in the `tools/petscfem.el` Emacs Lisp file. Another mode that can serve for colorization is the *"shell-script"* mode. In order to associate the `.epl` or `.depl` extensions to this mode, add this to your `~/.emacs` file

```
(setq auto-mode-alist
  (cons '("\\.\\(\\|d\\)epl$" . shell-script-mode) auto-mode-alist))
```

4.5.1 Installing PETSc-FEM mode

In order to use the mode, you have to copy the file `tools/petscfem.el` to somewhere accessible for Emacs (`/usr/share/emacs/site-lisp` is a good candidate). There is also a file `tools/petscfem-init.el` that contains some basic configuration, you can also copy it, or insert directly its contents into your `.emacs`.

```
;; Load basic PETSc-FEM mode
(load-file "/path/to/petscfem/tools/petscfem.el")

;; Load 'petscfem-init' or insert directly its contents
;; below and configure
(load-file "/path/to/petscfem/tools/petscfem-init.el")
```

4.5.2 Copying and pasting PETSc-FEM options

PETSc-FEM modules have a lot of options, and is mandatory to have fast access to all of them and to their documentation. The HTML documentation has a list of all of them at the end of the user's guide. For easier access there is also an info file (`doc/options.info`) that has a page for each of the options. You can browse it with the standalone *GNU Info* program or within Emacs with the info commands. In the last case you have the additional advantage that you can very easily find the documentation for a given option with a couple of keystrokes and paste options from the manual into your PETSc-FEM data file.

For jumping to the documentation for an option, put the cursor on the option and press `C-h C-i` (that is `<Ctrl-h><Ctrl-i>`, this is the key-binding for `info-lookup-symbol`). You will get in the minibuffer something like

```
Describe symbol (default print_internal_loop_conv):
```

If you press `RET` then you jump to the corresponding page in the info file. From there you can browse the whole manual. Pressing `s` (`Info-search`) allows to search for text in the whole manual. When done, you can press `q` (`Info-exit`) or `x` (`my-Info-bury-and-kill` defined in `tools/petscfem.el`).

If you don't know exactly what option you are looking for, then you can search in the manual or launch `info-lookup-symbol` with the start of the command and then use *"completion"* to finish writing the option.

If you want to copy some option from the info manual to the data file, then you can use the usual keyboard or mouse *copy-and-paste* methods of Emacs. Also pressing `c` (`my-Info-lookup-copy-keyword`) in the info manual copies the name of the currently visited option to the *"kill-ring"*. Then, in the data file buffer press as usual, `C-y` (`yank`) to paste the last killed option. If you paste several options from the manual, then you can navigate between them by pasting the last with `C-y` and then going back and forth in the kill ring with `M-y` (`yank-forw`, usually `M-` stands for pressing the `<Alt>` or `<Escape>` key).

For more information, see the Emacs manual, specially the documentation for the `info` and `info-lookup` modes.

5 Text hash tables

In many cases, options are passed from the user data file to the program in the form of small databases which are called *"text hash tables"*. This consist of several lines, each one starting with a keyword and followed by one or several values. When reading this data in the `read_mesh()` routine, the program doesn't know anything about neither the keywords nor the appropriate values for these keywords. Just when the element is processed by the element module via the `assemble()` function call, for instance assembling residuals or matrices, the element routine written by the application writer reads values from this database and apply it to the elemset. The text hash table is stored internally as a correspondence between the keys and the values, both in the form of strings. The key is the first (space delimited) token and the value is the rest of the string, from the end of the space separator to the end of the line. Usually the values are strings like a "C" identifier (`chunk_size` for instance). As the values are stored in the form of strings, almost any kind of values may be stored their, for instance

```
non_linear_method Newton    # string value
max_iter 10                 # integer value
viscosity 1.2e-4            # double value
```

or lists and combinations of them. It's up to the application writer to decode this values at the element routine level. Several routines in the `getprop` package help in getting this (see §5.4).

Some of this options are used for internal use of the PETSc-FEM code, for instant `chunk_size` sets the quantity of elements that are processed by the elemset routine at each time.

In section §5.1, some specific properties of the elemset properties text hash table are described.

5.1 The elemset hash table of properties

A very common problem in FEM codes is how to pass element physical or numerical properties (conductivities, viscosities, etc...), to the element routine. In PETSc-FEM you can pass arbitrary *"per elemset"* quantities via an *"elemset properties hash table"*. This comes after the element header as in the following example

```
elemset nsi_tet 4 fat 4   # elemset header
props                     # per element properties (to be explained later)
#
# Elemset properties hash table
#
name My Mesh
geometry cartesian2d
ndim 2
npg 4
couple_velocity 0
weak_form 1
```

```
# physical properties
Dt .1           # time step
viscosity 1.
# next lines flags end of the properties hash table
__END_HASH__
# element connectivities
 1 3 4 2
 3 5 6 4
 5 7 8 6
...
< more element connectivities here >
...
 193 195 196 194
 195 197 198 196
 197 199 200 198
 199 201 202 200
# next lines flags end of elemset connectivities
__END_ELEMSET__
```

In this example we define several properties, containing doubles (`viscosity` and `Dt`), integers (`ndim`, `npg`, etc...) or strings (`name`). This hash table is stored in the form of an associative array (*"hash"*) with a text key, the first non-blank characters and a value composed of the rest of the line. In the previous example

$$\begin{aligned} \text{Key: } \texttt{"name"} &\rightarrow \text{Value: } \texttt{"My Mesh"} \\ \text{Key: } \texttt{"ndim"} &\rightarrow \text{Value: } \texttt{"2"} \\ \text{Key: } \texttt{"viscosity"} &\rightarrow \text{Value: } \texttt{"1."} \end{aligned} \tag{1}$$

and so on. This hash table is read in an object of type "`TextHashTable`" without checking whether this properties apply to the particular elemset or not. The values are stored strictly as string, so that no check is performed on the syntax of entering double values or integers.

5.2 Text hash table inclusion

Often, we have some physical or numerical parameter that is common to a set of elemsets, for instance gravity in shallow water, or viscosity in Navier-Stokes. In this case, these common properties can be put in a separate table with a `table ...` header, and included in the elemsets with `_table_include` directives. The `table ...` sections (not associated to a particular elemset) have to be put in preference before the calling elemset, for instance

```
table steel_properties
density 13.2
viscosity 0.001
<more steel properties here ...>
__END_HASH__
```

```
elemset nsi_tet 4
_table_include steel_properties
npg 8
weak_form 1
<more properties for this elemset here ...>
__END_HASH__
 1 3 4 2
<more element connectivities here ...>
```

Text hash tables may be recursively included to any depth. The text hash table for an elemset may be included in other elemset referencing it by its `name` entry, for instance

```
elemset nsi_tet 4
name volume_elemset_1
geometry cartesian2d
npg 4
viscosity 0.23
<more properties here ...>
__END_HASH__
1 3 4 2
<more connectivities here ...>
__END__ELEMSET__


elemset nsi_tet 4
name volume_elemset_2
_table_include volume_elemset_1 # includes properties from the
                                # previous elemset
__END_HASH__
 5 4 76 45
<more connectivities here ...>
__END__ELEMSET__
```

5.3 The global table

If one table is called `global_options` then all other hash tables inherit their properties, without needing to explicitly include it. For instance in this case

```
table global_options
viscosity 0.023
tau_fac 0.5
__END_HASH__

elemset nsi_tet 4
name elemset_1
```

```
__END_HASH__
 5 4 76 45
<more connectivities here ...>
__END__ELEMSET__
```

the elemset `elemset_1` gets a value for viscosity from the global hash table of 0.023. The `global_options` table may include other tables as well.

Note: In previous versions of the code, the `table` keyword may be omitted for the `global_options` table. For instance, the previous example can be entered as

```
global_options
viscosity 0.023
tau_fac 0.5
__END_HASH__
```

This usage is obsolete, and will be deprecated.

5.4 Reading strings directly from the hash table

Once inside the element routine values can be retrieved with routines from the `TextHashTable` class, typically `get_entry`, for instance

```
char *geom;
thash->get_entry("geometry",geom);
```

this returns a pointer to the internal value string "`geom`" (this is documented with the `TextHashTable` class. You can then read values from it with routines from `stdio` (`sscanf` and friends). You should not try to modify this strings, for instance with the `strtok()` function. In that case, copy the string in a new fresh string (remember to delete it after to avoid memory leak). In this way you can pass arbitrary information (strings, integer, doubles) to the element routine.

5.5 Reading with 'get_int' and 'get_double'

As most of the time element properties are either of integer or double type, two specific routines are provided "`get_int`" and "`get_double`", for instance

```
ierr = get_int(thash,"npg",&npg);
```

where the integer value is directly returned in "`npg`". You can specify also a default value and read several values at once.

5.6 Per element properties table

Many applications need a mechanism for storing values per element, for instance when dealing with physical properties variable varying spatially in a continuous way. If the properties is piecewise constant, then you can define an elemset for each constant patch, but if it varies continuously, then you should need an elemset for each element, which

conspires with efficiency. We provide a mechanism to pass "per element" values to the element routine. At the moment this is only possible for doubles. These properties are specified in the same line of the connectivities, for instance

```
elemset nsi_tet 4         # elemset header
props   cond  cp  emiss   # name of properties to be defined "per element"
# Elemset properties hash table
geometry cartesian2d
ndim 2
npg 4
# physical properties
Dt .1           # time step
viscosity 1.
# next lines flags end of the properties hash table
__END_HASH__
# element connectivities, physical properties per element
 1 3 4 2  1.1 2.3 0.7
 3 5 6 4  1.2 2.1 0.8
 5 7 8 6  1.3 2.2 0.9
...
< more element connectivities and physical props. here >
...
 193 195 196 194  1.5 2.0 0.8
 195 197 198 196  1.6 2.1 0.7
 197 199 200 198  1.7 2.2 0.6
 199 201 202 200  1.8 2.3 0.5
# next lines flags end of elemset connectivities
__END_ELEMSET__
```

Here we define that properties "`cond`", "`cp`" and "`emiss`" are to be defined per element. We add to each element connectivity line the three properties. There are two ways to access these properties. The corresponding values are stored in an array "`elemprops`" of length $n_{\text{props}} \times n_{\text{elem}}$, where n_{props} is the number of "per element" properties (3 in this example) and n_{elem} is the number of elements in the mesh. Also there is a macro "`ELEMPROPS(k,iprop)`" that allows treating it as a matrix. So, you can access this values with

```
cond = ELEMPROPS(k,0);  // conductivity of element k
cp   = ELEMPROPS(k,1);  // specific heat  of element k
emiss= ELEMPROPS(k,2);  // emissivity of element k
```

5.7 Taking values transparently from hash table or per element table

With the tools described so far you can access constant properties on one hand (the same for the whole elemset) and per element properties. Now, given a property, you should decide whether it should be assumed to be constant for all the elemset or whether it can be taken "per element". The second is the more general case, but to take all the possible

properties as "per element' may be too much core memory consuming. There is then a mechanism to allow the application writer to get physical properties without bothering of whether the user has set them in the properties hash table or in the per-element properties table.

First, the application writer reserves an array of doubles large enough to contain all the needed properties. (This doesn't scale with mesh size so you can be generous here, or either use dynamic memory allocation). Before entering the element loop, the macro "`DEFPROP(prop_name)`" determines whether the property has been passed by one or the other of the mechanisms. This information is stored in an integer vector "`elprpsindx[MAXPROP]`". Also, it assigns a position in array "`propel`" so that "`propel[prop_name_indx]`" contains the given property. Then, once inside the element loop a call to the function "`load_props`" loads the appropriate values on "`propel[MAXPROP]`", independently how thay have been defined. A typical call sequence is as follows

```
// Maximum number of properties to be loaded via load_prop
#define MAXPROP 100
int iprop=0, elprpsindx[MAXPROP]; double propel[MAXPROP];

// determine which mechanism passes 'conductivity'
DEFPROP(conductivity)

// conductivity is found (after calling load_props()) in
// propel[conductivity_indx]
#define COND (propel[conductivity_indx])

// Other properties
DEFPROP(propa)
#define PROPA (propel[propa_indx])

DEFPROP(propb)
#define PROPB (propel[propb_indx])

DEFPROP(propc)
#define PROPC (propel[propc_indx])

DEFPROP(propp)
#define PROPP (propel[propp_indx])

DEFPROP(propq)
#define PROPQ (propel[propq_indx])

// Total number of properties loaded
int nprops=iprop;

// Set error if maximum number of properties exceeded
```

```
assert(nprops<=MAXPROP);

// ... code ...

// loop over elements
for (int k=el_start; k<=el_last; k++) {
    // check if this element is to be processed
    if (!compute_this_elem(k,this,myrank,iter_mode)) continue;

    // Load properties either from properties hash table or from
    // per element properties table
    load_props(propel,elprpsindx,nprops,&(ELEMPROPS(k,0)));

    // ... more code ...

    // use physical element property
    double veccontr += wpgdet * COND * dshapex.t() * dshapex;

    // ...
```

First, we allocate for 100 entries in "`elprpsindx`" and "`propel`" arrays, and set the counter "`prop`" to 0. Then we call "`DEFPROP`" for properties "`conductivity`" and "`propa`" thru "`propq`". After this, we check that the maximum number of properties to be defined is not exceeded and enter the element loop. After checking, as usual, if the element needs processing, we call "`load_props`" in order to effectively load element properties in propel and, after this, we can use them as "`propel[conductivity_indx]`" and so on. Macro shortcuts "`COND`", "`PROPA`" are handy for this.

6 The general advective elemset

6.1 Introduction to advective systems of equations

Advective system of equations are of the form

$$\frac{\partial U}{\partial t} + \frac{\partial \mathcal{F}_i(U)}{\partial x_i} = G \tag{2}$$

where U is the *"state vector"* of the fluid in *"conservative variables"*. Examples of these are the inviscid fluid equations (the *"Euler"* or *"gas dynamic equations"*, the *"shallow water equations"*, and scalar advective systems that represents the transport of a scalar property (like temperature or concentration of a component) by a moving fluid.

The conservative variables for the Euler equations are

$$U = \begin{bmatrix} \rho \\ \rho \mathbf{u} \\ \rho e \end{bmatrix} \tag{3}$$

for the Euler equations. In general, U is a vector of n_{dof} components. $\mathcal{F}_i(U)$ is the *"flux vector"*. t can be thought as a matrix of $n_{\text{dof}} \times n_{\text{dim}}$ components. The row index

corresponds to a field value, whereas the column index is a spatial dimension. G is a source vector. For the 2D or 3D Euler equations it is null, but if we consider 1D flow in a tube of varying section, then it has a source term in the momentum equation, due to the reaction on the wall. Also, for the shallow water equations, there is a reaction term in the momentum balance equations if there is a varying bathymetry.

The relation of the flux vector with the state vector is the heart of the advective system. In fact, the discretization of advective systems may be put in a completely abstract setting, where the unique thing that varies from one system to another is the definition of the flux function itself. The discretization of advective systems in PETSc-FEM has been done in this way, so that it is easy to add other advective systems by only adding the new flux function.

Applying the chain rule, and noting that the fluxes only depend on position through their dependence on the state vector, we arrive to

$$\frac{\partial F_i}{\partial x_i} = A_i \, \frac{\partial U}{\partial x_i} \tag{4}$$

where

$$A_i = \frac{\partial F_i}{\partial U} \tag{5}$$

are the *"jacobians of the advective fluxes"*.

6.2 Discretization of advective systems

Using the Finite Element Method, with weight functions $W_j(\mathbf{x})$ and interpolation functions $N_j(\mathbf{x})$ results in

$$\mathbf{M}\dot{\mathbf{U}} + \mathbf{F}(\mathbf{U}) = \mathbf{G} \tag{6}$$

where

$$\mathbf{U} = \begin{bmatrix} U_1 \\ U_2 \\ \vdots \\ U_{N_{\text{nod}}} \end{bmatrix} \tag{7}$$

so that $\mathbf{U}$ has $N_{\text{d.o.f.}} = n_{\text{dof}} \times N_{\text{nod}}$ components.

$\mathbf{M}$ (of dimension $N_{\text{d.o.f.}} \times N_{\text{d.o.f.}}$) is the mass matrix. If we look at it as an $N_{\text{nod}} \times N_{\text{nod}}$ block matrix with blocks of size $n_{\text{dof}} \times n_{\text{dof}}$, then the i, j block is

$$\mathbf{M}_{ij} = \int_\Omega N_i(\mathbf{x}) \, N_j(\mathbf{x}) \, \mathrm{d}\mathbf{x} \tag{8}$$

the i block of the global flux and source vector $\mathbf{F}$ and $\mathbf{G}$ are

$$\begin{aligned} \mathbf{F}_i &= \int_\Omega N_i(\mathbf{x}) \, \frac{\partial F_k}{\partial x_k} \, \mathrm{d}\mathbf{x} \\ \mathbf{G}_i &= \int_\Omega N_i(\mathbf{x}) \, G(\mathbf{x}) \, \mathrm{d}\mathbf{x} \end{aligned} \tag{9}$$

If the flux vector term is integrated by parts then we have the *"weak form"*

$$\mathbf{F}_i = -\int_\Omega \frac{\partial N_i}{\partial x_k} \, F_k \, \mathrm{d}\mathbf{x} + \int_\Gamma n_k F_k(\mathbf{x}) \, \mathrm{d}\mathbf{x} \tag{10}$$

where Γ is the boundary of Ω. This formulation is the *"Galerkin"* or *"centered"* one. It is equivalent to approximate first derivatives by centered differences in the Finite Difference Method. It is well known that the Galerkin formulation leads to oscillations for advective systems, and this is solved by adding a *"stabilizing term"* to the discretized equations.

6.3 SUPG stabilization

In the SUPG (for *"Streamline Upwind/Petrov Galerkin"*) formulation of Hughes et.al. The stabilized formulation is

$$(\mathbf{M}\dot{\mathbf{U}} + \mathbf{F}(\mathbf{U}) - \mathbf{G})_j + \sum_e \int_{\Omega_e} (P_{\text{SUPG}})^e_j \, (\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial x_i} - G) = 0 \tag{11}$$

where the whole expression corresponds to the j-th block of size n_{dof} in the global equations. Note that, as the added term is a *"weighted residual"* form of the residual (the term in parentheses), then the continuum solution is solution of these discrete equations – we say that this is a *"consistent formulation"*. P_{SUPG} is a matrix of $n_{\text{dof}} \times n_{\text{dof}}$ the SUPG *"perturbation function"*, usually defined as

$$(P_{\text{SUPG}})^e_j = \boldsymbol{\tau}^e \, A_j \frac{\partial N}{\partial x_j} \tag{12}$$

where $\boldsymbol{\tau}^e$ are the *"characteristic"* or *"intrinsic time"* of the element, defined as

$$\boldsymbol{\tau}^e = \frac{h^e}{||\mathbf{A}||} \tag{13}$$

where h^e is the size of the element and and $||\mathbf{A}||$ represents some norm of the vector of jacobians. There is a variety of possibilities for computing both quantities. For instance h^e may be computed as the largest side of the element, or as the radius of the circle with the same area. On the other hand, $||\mathbf{A}||$ may be computed as the maximum eigenvalue of all the linear combinations of the form $n_j A_j$, with n_j a unit vector, i.e. the maximum propagation velocity possible in the fluid, that is

$$||\mathbf{A}|| = \max_{j=1,\ldots,n_{\text{dim}}} \max_{k=1,\ldots,n_{\text{dof}}} |\lambda^j_k| \tag{14}$$

where λ^j_k is the k-th eigenvalue of jacobian A_j. For the Euler equations, it turns out to be that this corresponds to pressure waves propagating in the direction of the fluid and is $c + u$ where c is the speed of sound and u the absolute value of velocity. For the shallow water equations, its value is $u + \sqrt{gh}$ where g is gravity acceleration and h the local water elevation with respect to bottom.

6.4 Shock capturing

For problems with strong shocks, (shock waves in Euler, or hydraulic jumps in shallow water) the standard SUPG stabilizing term may not be sufficient. Then an additional stabilizing term is added, so that the stabilized equations are now of the form

$$\begin{aligned}(\mathbf{M}\dot{\mathbf{U}} + \mathbf{F}(\mathbf{U}) - \mathbf{G})_j + \sum_e \int_{\Omega_e} (P_{\text{SUPG}})^e_j \, (\frac{\partial U}{\partial t} + \frac{\partial F_i}{\partial x_i} - G) + \\ + \sum_e \int_{\Omega_e} \delta_{\text{sc}} \, \frac{\partial N_j}{\partial x_i} \, \frac{\partial U}{\partial x_i} = 0\end{aligned} \tag{15}$$

Note that, in contrast with the SUPG term, the new, so-called shock capturing term is no more *"consistent"*. δ_{sc} is a scalar – the so called *"shock capturing parameter"*. Often, when shock capturing is added, we diminish the amount of stabilization in the SUPG term in order to compensate and not to have an over-diffusive scheme. We will not enter in the details of this computations, refer to [2] for further details.

6.5 Creating a new advective system

New advective systems may be added to PETSc-FEM only by defining their flux function, jacobians and other quantities. This means that you don't need to code details of the numerical discretization. Follow these steps

1. Create the flux function in a file by itself, in the `applications/advective` directory. (The better is to start copying one of the existing advective systems, for instance `ffeuler.cpp` or `ffshallw.cpp`.) The arguments to flux function routines is described in section 6.6. The name of the function has to be of the form `flux_fun_<system>` where `<system>` identifies the new system. We assume that you write the flux function `flux_fun_new_adv_sys` in file `applications/advective/ffnadvs.cpp`.

2. Add the file in the `MYOBJS` variable in the Makefile, for instance

   ```
   MYOBJS = advective.o adv.o absorb.o ffeuler.o \
   ffshallw.o ffadvec.o
   ```

3. Define the new derived classes `volume_new_adv_sys` and `absorb_new_adv_sys` by adding a line at the end of the file `applications/advective/advective.h` as in the following example.

   ```
   // Add here declarations for further advective elemsets.
   /// Euler equations for inviscid (Gas dynamics eqs.)
   ADVECTIVE_ELEMSET(euler);
   /// Shallow water equations.
   ADVECTIVE_ELEMSET(shallow);
   /// Advection of multiple scalar fields with a velocity field.
   ADVECTIVE_ELEMSET(advec);
   /// My new advective system
   ADVECTIVE_ELEMSET(new_adv_sys);             // <- Add this line.
   ```

4. Recompile.

6.6 Flux function routine arguments

Currently, the interface is the following

```
typedef int FluxFunction(const RowVector &U,
        int ndim,const Matrix &iJaco,
Matrix &H, Matrix &grad_H,
Matrix &flux,vector<Matrix *> A_jac,
Matrix &A_grad_U, Matrix &grad_U,
Matrix &G_source,Matrix &tau_supg, double &delta_sc,
double &lam_max,
TextHashTable *thash,double *propel,
void *user_data,int options)
```

(This may eventually change – in any case, if you are interested in adding a new advective system, then see the actual description in the `advective.h` file in the distribution.) The meaning of these arguments are listed below. When the size is specified, it means that the argument is a Newmat or FastMat matrix.

In some situations the flux function routine must compute only some of the required values. For instance, when computing the contribution of the absorbing boundary elements there is no need to compute the parameters regarding stabilizing terms. This is controlled with the parameter `options` which can take the values `DEFAULT`, `COMP_UPWIND` and `COMP_SOURCE`. In the list below, it is indicated under which conditions the specific quantity must be computed.

- `const RowVector &U` (input, size n_{dof}) This is the state vector – you must return the flux, jacobians and other quantities for *this* state vector.
- `int ndim` (input) The dimension of the space.
- `const Matrix &iJaco` (input, size $n_{\text{dim}} \times n_{\text{dim}}$) The jacobian of the master to element coordinates in the actual gauss points. This may be used in order to calculate the characteristic size of the element
- `Matrix &H` (input, size $1 \times n_H$) In the shallow water, the source term G depends on the gradient of the depth $H(x)$, and in the 1D Euler equations, on the area of the tube section $A(x)$. This is taken into account by PETSc-FEM by assuming that the user enters in the `nodedata` section `nu` $= n_{\text{dim}} + n_H$ quantities per node, where the first n_{dim} quantities are the node coordinates and the rest are assumed that are node data that has to be passed to the flux function routine (together with its gradient) in order to compute the source term.
- `Matrix &grad_H` (input, size $n_{\text{dim}} \times n_H$) the gradient of the quantities in H (see previous entry).
- `Matrix &flux` (output, size $n_{\text{dof}} \times n_{\text{dim}}$) Each column is the vector F_j of fluxes for each of the governing equations.
- `vector<Matrix *> A_jac` (output, size: a vector of n_{dim} pointers to matrices of $n_{\text{dof}} \times n_{\text{dof}}$). Each matrix is the jacobian matrix as defined by (5). To access the `jd` jacobian matrix you may write `(*A_jac[(jd)-1])`. The macro `AJAC(jd)`, defined in `advective.h` expands to this.

- `Matrix &A_grad_U` (output, size $n_{\text{dof}} \times 1$, compute if `options & COMP_UPWIND`) This is the accumulation term $A_k \left(\partial U / \partial x_j\right) = \frac{\partial F_j}{\partial x_j}$.
- `Matrix &grad_U` (input, size $n_{\text{dim}} \times n_{\text{dof}}$) The gradient of the state vector.
- `Matrix &tau_supg` (output, size: either 1×1 or $n_{\text{dof}} \times n_{\text{dof}}$, compute if `options & COMP_UPWIND`). This is the $\boldsymbol{\tau}$ intrinsic time scale – it may be either a scalar or a matrix. Beware that even in the case where it is a scalar it must be returned as a Newmat `Matrix` object of dimensions 1×1.
- `double &delta_sc` (output, double, compute if `options & COMP_UPWIND`) This is the *"shock capturing"* parameter as described in 6.4. Set to 0 if no shock capturing is added.
- `double &lam_max` (output, double, compute if `options & COMP_UPWIND`) The maximum propagation speed. The expression is (14) but normally it may be computed directly from the state vector. This is used to compute upwind, and also the automatic and local time step.
- `TextHashTable *thash` (input) This is the `TextHashTable` of the elemset. Physical and numerical parameters can be passed from the user input data file to the flux function routine through this (specific heat, specific heat ratio, gravity... for instance). Beware that the flux function routine is called at each Gauss point, and decoding of the table may be time expensive, so that if the properties are constant over all the mesh you can decode them once and leave the decoded data in static variables.
- `double *propel` (input, double array, compute if `options & COMP_UPWIND`) This is the table of per-element properties. Physical and numerical parameters that are not constant for all the elemsets, can be passed from the user input data file to the flux function routine through this.
- `void *user_data` (input). Arbitrary information may be passed from the main routine to the flux function through this pointer.
- `int options` (input, integer).
- `Matrix &G_source` (output, size $n_{\text{dof}} \times 1$, compute if `options & COMP_SOURCE`) The source vector.

6.6.1 Options

General options:

- `double Courant` (default=`0.6`):
  The Courant number. (found in file: `adv.cpp`)
- `double Dt` (default=`0.`):
  Time step. (found in file: `adv.cpp`)

- `double atol` (default=`1e-6`):
  Absolute tolerance when solving a consistent matrix (found in file: `adv.cpp`)
- `int auto_time_step` (default=`1`):
  Chooses automatically the time step from the selected Courant number (found in file: `adv.cpp`)
- `int consistent_supg_matrix` (default=`0`):
  Uses consistent SUPG matrix for the temporal term or not. (found in file: `adv.cpp`)
- `double dtol` (default=`1e+3`):
  Divergence tolerance when solving a consistent matrix (found in file: `adv.cpp`)

- `int local_time_step` (default=`1`):
  Chooses a time step that varies locally. (Only makes sense when looking for steady state solutions. (found in file: `adv.cpp`)
- `int maxits` (default=`150`):
  Maximum iterations when solving a consistent matrix (found in file: `adv.cpp`)
- `int measure_performance` (default=`0`):
  Measure performance of the `comp_mat_res` jobinfo. (found in file: `adv.cpp`)
- `int nfile` (default=`1`):
  Sets the number of files in the "rotary save" mechanism. (see 7.2) (found in file: `adv.cpp`)
- `int nrec` (default=`1000000`):
  Sets the number of states saved in a given file in the "rotary save" mechanism (see 7.2 (found in file: `adv.cpp`)
- `int nsave` (default=`10`):
  Save state vector frequency (in steps) (found in file: `adv.cpp`)
- `int nsaverot` (default=`100`):
  Save state vector frequency with the "rotary save" mechanism. (see 7.2) (found in file: `adv.cpp`)
- `int nstep` (default=`10000`):
  The number of time steps. (found in file: `adv.cpp`)
- `int nstep_cpu_stat` (default=`10`):
  Output CPU time statistics for frequency in time steps. (found in file: `adv.cpp`)

- `int print_internal_loop_conv` (default=`0`):
  Prints the convergence history when solving a consistent matrix (found in file: `adv.cpp`)
- `int print_linear_system_and_stop` (default=`0`):
  After computing the linear system prints Jacobian and right hand side and stops.. (found in file: `adv.cpp`)

- `double rtol` (default=`1e-3`):
  Relative tolerance when solving a consistent matrix (found in file: `adv.cpp`)
- `string save_file` (default=`outvector.out`):
  Filename for saving the state vector. (found in file: `adv.cpp`)
- `string save_file_pattern` (default=`outvector%d.out`):
  The pattern to generate the file name to save in for the rotary save mechanism. (found in file: `adv.cpp`)
- `double start_time` (default=`0.`):
  Counts time from here. (found in file: `adv.cpp`)
- `double tol_mass` (default=`1e-3`):
  Tolerance when solving with the mass matrix. (found in file: `adv.cpp`)

Generic elemset "`advecfm2`":

- `double beta_supg` (default=`0.8`):
  Parameter to control the amount of SUPG perturbation added to the mass matrix to be consistent SUPG `beta_supg` =0 implies consistent Galerkin and `beta_supg` =1 implies full consistent SUPG. (found in file: `advecfm2.cpp`)
- `string geometry` (default=`cartesian2d`):
  Type of element geometry to define Gauss Point data (found in file: `advecfm2.cpp`)
- `int lumped_mass` (default=`1`):
  Use lumped mass. (found in file: `advecfm2.cpp`)
- `int weak_form` (default=`1`):
  Use the weak form for the Galerkin part of the advective term. (found in file: `advecfm2.cpp`)

Flux function "`ffeulerfm2`": Euler eqs.

- `double gamma` (default=`1.4`):
  The specific heat ratio. (found in file: `ffeulerfm2.cpp`)
- `int shock_capturing` (default=`0`):
  Add shock-capturing term. (found in file: `ffeulerfm2.cpp`)
- `double shock_capturing_threshold` (default=`0.1`):
  Add shock-capturing term if relative variation of variables inside the element exceeds this. (found in file: `ffeulerfm2.cpp`)
- `double tau_fac` (default=`1.`):
  Scale the SUPG upwind term. (found in file: `ffeulerfm2.cpp`)

Flux function "`ffswfm2`": Shallow water eqs.

- `double gravity` (default=`1.`):
  Acceleration of gravity (found in file: `ffswfm2.cpp`)

- `int shock_capturing` (default=`0`):
  Add shock-capturing term. (found in file: `ffswfm2.cpp`)
- `double shock_capturing_threshold` (default=`0.1`):
  Add shock-capturing term if relative variation of variables inside the element exceeds this. (found in file: `ffswfm2.cpp`)
- `double tau_fac` (default=`1.`):
  Scale the SUPG upwind term. (found in file: `ffswfm2.cpp`)

6.7 The hydrology module

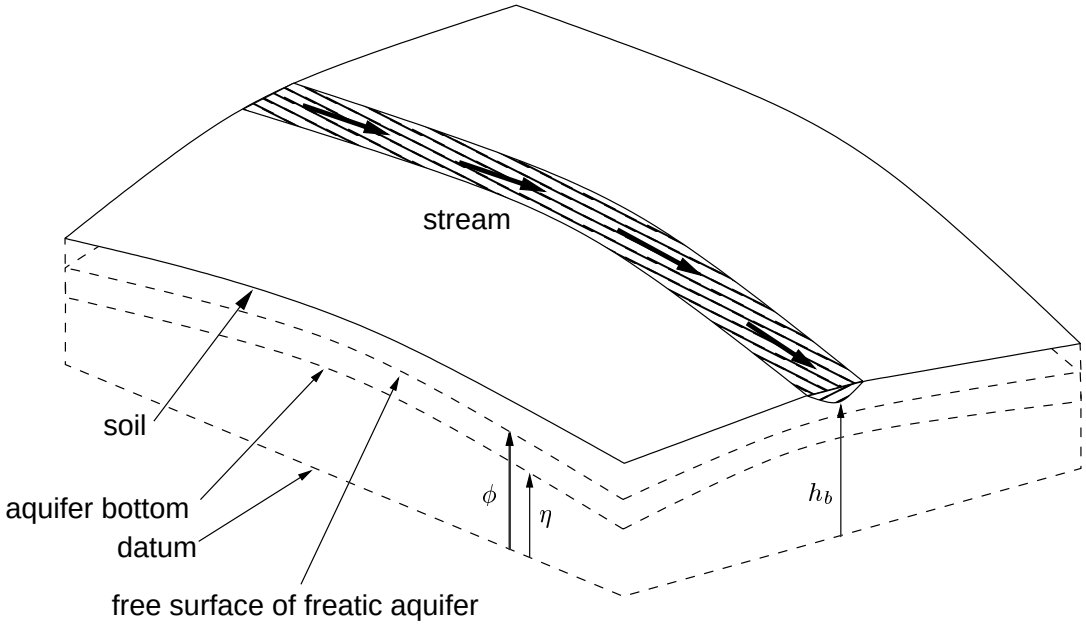


Figure 2: Aquifer/stream system.

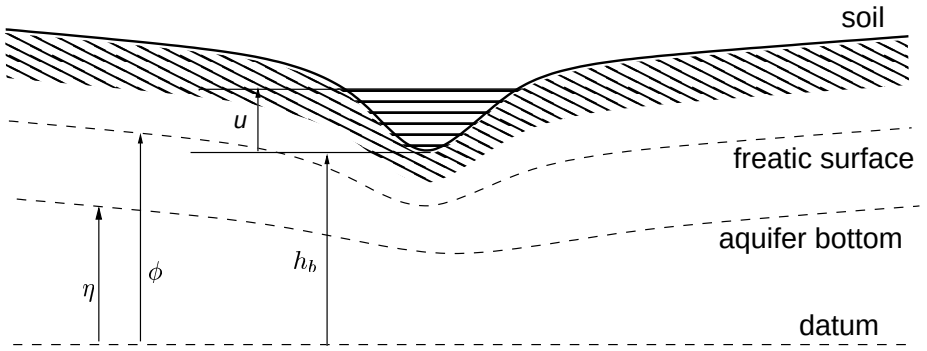


Figure 3: Aquifer/stream system. Transverse 2D view

This module solves the problem of subsurface flow in a free aquifer, coupled with a surface net of 1D streams. To model such system three elemsets must be used: an `aquifer`

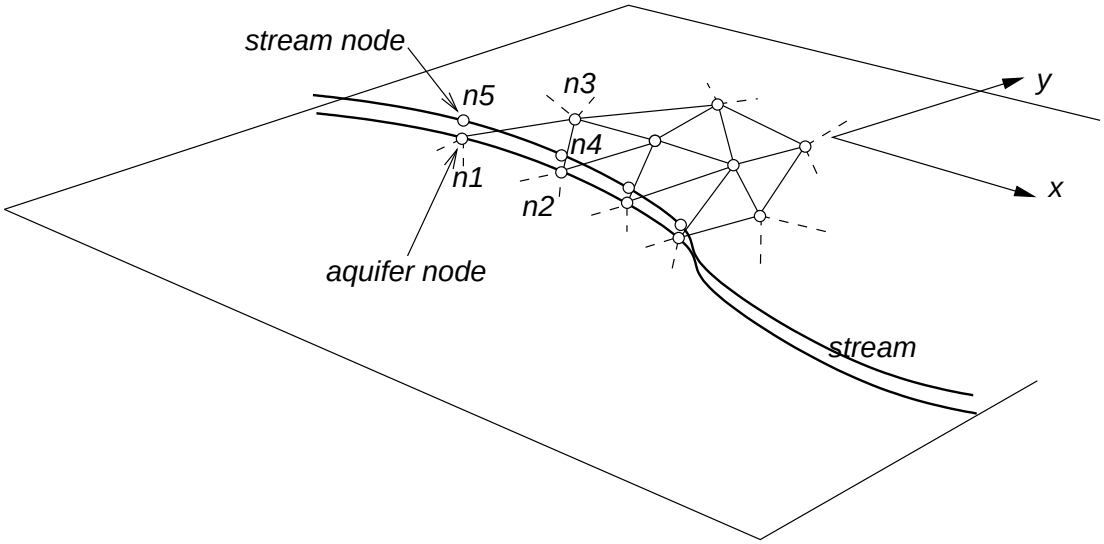


Figure 4: Aquifer/stream system. Discretization.

system representing the subsurface aquifer, a `stream` elemset representing the 1D stream and a `stream_loss` elemset representing the losses from the stream to the aquifer (or vice versa) see figures 2 and 3.

The `aquifer` elemset is a 2D elemset with triangle or quadrangle elements (see figure 4). A per-element property `eta` represents the height of the aquifer bottom to a given datum. The corresponding unknown for each node is the piezometric height or the level of the freatic surface at that point ϕ. On the other hand, the `stream` elemset represents a 1D stream of water. It has its own nodes, separate from the aquifer nodes, whose coordinates must coincide with some corresponding node in the aquifer. For instance, the triangular aquifer element in the figure is connected to nodes $n1$, $n2$ and $n3$, while the stream element is connected to nodes $n4$ and $n5$. $n1$ and $n5$ have the same coordinates (but different unknowns) and also $n2$ and $n4$. A node constant field (so called "H-fields") represents the stream bottom height h_b, with reference to the datum. So that, normally, we have for each node two coordinates and the stream bottom height. (`ndim=2 nu=3 ndof=1`). The unknown for these nodes is the height u of the stream free water surface with reference to the stream bottom. The channel shape and friction model and coefficients are entered via properties described below. If the stream level is above the freatic aquifer level ($h_b+u > \phi$) then the stream losses water to the aquifer and vice versa.

The equation for the aquifer integrated in the verical direction is

$$\frac{\partial}{\partial t}\left(S(\phi-\eta)\phi\right) = \nabla\cdot\left(K(\phi-\eta)\nabla\phi\right) + \sum G_a \tag{16}$$

where S is the storativity and G is a source term, due to rain, losses from streams or other aquifers.

The equation for the stream is, according to the *"Kinematic Wave Model" KWM* approach,

$$\frac{\partial A(u)}{\partial t} + \frac{\partial Q(A(u))}{\partial s} = G_s \tag{17}$$

Where u is the unknown field that represents the height of the water in the channel with respect to the channel bottom as a function of time and a linear arc coordinate along the stream, A is the transverse cross section of the stream and depends, through the geometry of the channel, on the channel water height u. Q is the flow rate and, under the KWM model is a function only of A through the friction law.

$$Q = \gamma A^m \tag{18}$$

where $\gamma = C_h\, S^{1/2}\, P^{-1}$ and $m = 3/2$ for the Chèzy friction model, and $\gamma = \bar{a}\, n^{-1}\, S^{1/2}\, P^{-2/3}$ and $m = 5/3$ for the Manning model, where $S = (\mathrm{d}h_b/\mathrm{d}s)$ is the slope of the stream bottom, P is the wetted perimeter, and C_h, $\bar{a}$ and n are model constants. G_s represent the gain or loss of the stream, and the main component is the loss to the aquifer

$$G_s = P/R_f(\phi - h_b - u) \tag{19}$$

where R_f is the resistivity factor per unit arc length of the perimeter. The corresponding gain to the aquifer is

$$G_a = -G_s\, \delta_{\Gamma_s} \tag{20}$$

where Γ_s represents the planar curve of the stream and δ_{Γ_s} is a Dirac's delta distribution with a unit intensity per unit length, i.e.

$$\int f(\mathbf{x})\, \delta_{\Gamma_s}\, \mathrm{d}\Sigma = \int_0^L f(\mathbf{x}(s))\, \mathrm{d}s \tag{21}$$

The `stream_loss` elemset represents this loss, and a typical discretization is shown in figure 4. The stream loss element is connected to two nodes on the stream and two on the aquifer and must be entered in that order in the element connectivity table, for instance

```
elemset stream_loss 4
<.... elemset properties...>
__END__HASH__
...
<n5> <n4> <n1> <n2>
...
__END__ELEMSET__
```

6.7.1 Related Options

- `double a_bar` (default=`1.`):

  Unit conversion factor for Manning friction law. (found in file:)

- `double B1` (default=`<required>`):

  Width of the channel (found in file:)

- `double Ch` (default=`<required>`):

  Chezy roughness coefficient (found in file:)

- `double diameter` (default=`<required>`):

  geometry of the channel (found in file:)

- `string friction_law` (default=`string("undefined")`):

  Choose friction law, may be `manning` or `chezy` (found in file:)

- `int impermeable` (default=`0`):

  Flag whether the element is impermeable ($R_f \to \infty$) or not. (found in file:)

- `double radius` (default=`<required>`):

  Radius of the channel (found in file:)

- `double Rf` (default= `1.`): Resistivity (including perimeter) of the stream to loss to the aquifer. (found in file:)

- `double roughness` (default=`1.`):

  Roughness coefficient for the Manning formula (a.k.a. n) (found in file:)

- `string shape` (default=`string("undefined")`):

  Choose channel section shape, may be `circular`, `rectangular` or `triangular`. (found in file:)

- `double wall_angle` (default=`<required>`):

  Width and height of the channel (found in file:)

- `double wall_angle` (default=`<required>`):

  Aperture angle of channel (found in file:)

- `double width` (default=`<required>`):

  Width of the channel (found in file:)

- `double width_bottom` (default=`<required>`):

  Width of bottom of channel (found in file:)

6.8 The Hydrological Model (cont.).

The implemented code solves the problem of subsurface flow in a free aquifer, coupled with a surface net of 2D or 1D streams (*"2D Saint-Venant Model"*, 2DSVM, *"1D Saint-Venant Model"*, 1DSVM, and *"Kinematic Wave model"*, KWM). To model such system three element sets must be used: an *aquifer* system representing the subsurface aquifer, a 2D or 1D (depending on the chosen model) *stream* element set representing the stream and a 2D or 1D *stream loss* element set representing the losses from the stream to the aquifer (or vice versa).

6.9 Subsurface Flow.

The *aquifer* element set is 2D linear triangle or quadrangle elements. A per-node property η represents the height of the aquifer bottom to a given datum. The corresponding unknown for each node is the piezometric height or the level of the freatic surface at that point ϕ. The equation for the aquifer integrated in the vertical direction is

$$\frac{\partial}{\partial t}\left(S(\phi-\eta)\phi\right) = \nabla\cdot\left(K(\phi-\eta)\nabla\phi\right) + \sum G_a, \quad \text{on } \Omega_{aq}\times(0,t], \tag{22}$$

where Ω_{aq} is the aquifer domain, S is the storativity, K is the hydraulic conductivity and G_a is a source term, due to rain, losses from streams or other aquifers.

6.10 Surface Flow.

6.10.1 2D Saint-Venant Model.

The *stream* element set represents a 2D or 1D stream of water. It has its own nodes, separated from the aquifer nodes, whose coordinates must coincide with some corresponding node in the aquifer. A constant per node field represents the stream bottom height h_b, with reference to the datum. That is why, normally, we have two coordinates and the stream bottom height for each node.
The equations for the 2D Saint-Venant open channel flow are the well known mass and momentum conservation equations integrated in the vertical direction. If we write this equations in the conservation matrix form, we have

$$\frac{\partial \mathbf{U}}{\partial t} + \frac{\partial \mathbf{F}_i(\mathbf{U})}{\partial x_i} = \mathbf{G}_i(\mathbf{U}), \qquad i=1,..,3, \qquad \text{on } \Omega_{st}\times(0,t], \tag{23}$$

where Ω_{st} is the stream domain, $\mathbf{U}=(h,hw,hv)^T$ is the state vector and the advective flux functions in eq. (23) are

$$\begin{aligned}\mathbf{F}_1(\mathbf{U}) &= (hw, hw^2+g\frac{h^2}{2}, hwv)^T,\\ \mathbf{F}_2(\mathbf{U}) &= (hv, hwv, hv^2+g\frac{h^2}{2})^T,\end{aligned} \tag{24}$$

where h is the height of the water in the channel with respect to the channel bottom, $\mathbf{u}=(w,v)^T$ is the velocity vector and g is the acceleration due to gravity. As in eq. (22), G_s represents the gain (or loss) of the river, the source term is

$$\mathbf{G}(\mathbf{U}) = (G_s,\ gh(S_{0x}-S_{fx})+f_chv+C_f\varpi_x|\varpi|,\ gh(S_{0y}-S_{fy})-f_chw+C_f\varpi_y|\varpi|)^T \tag{25}$$

where S_0 is the bottom slope and S_f is the slope friction.

$$\begin{aligned} S_{fx} &= \frac{1}{C_h h}w|\bar{u}|, \quad S_{fy} = \frac{1}{C_h h}v|\bar{u}| \quad \text{Chèzy model.}\\ S_{fx} &= \frac{n^2}{h^{4/3}}w|\bar{u}|, \quad S_{fy} = \frac{n^2}{h^{4/3}}v|\bar{u}|, \quad \text{Manning model.}\end{aligned} \tag{26}$$

where C_h and n (the Manning roughness) are model constants. Generally, the effect of coriolis force, related to the coriolis factor f_c, must be taken in account in the case of great

lakes, wide rivers and estuaries. The coriolis factor is given by $f_c = 2\omega \sin\psi$, where ω is the rotational rate of the earth and ψ is the latitude of the area under study. The free surface stresses in eq. (25) are expressed as the product between a friction coefficient and a quadratic form of the wind velocity, $\varpi(\varpi_x, \varpi_y)$, and

$$C_f = c_\varpi \frac{\rho_{air}}{\rho}, \tag{27}$$

where,

$$\begin{aligned} c_\varpi &= 1.25\times10^{-3}\varpi^{-1/5} \quad \text{if } |\varpi| < 1 \text{ m/s}, \\ c_\varpi &= 0.5\times10^{-3}\varpi^{1/2} \quad \text{if } 1 \text{ m/s} \le |\varpi| < 15 \text{ m/s}, \\ c_\varpi &= 2.6\times10^{-3} \quad \text{if } |\varpi| \ge 15 \text{ m/s}, \end{aligned} \tag{28}$$

6.10.2 1D Saint-Venant Model.

When velocity variations on the channel cross section are neglected, the flow can be treated as one dimensional. The equations of mass and momentum conservation on a variable cross sectional stream (in conservation form) are,

$$\begin{aligned} \frac{\partial \mathbf{A}(s,t)}{\partial t} + \frac{\partial \mathbf{Q}(\mathbf{A}(s,t))}{\partial s} &= G_s(s,t), \\ \frac{1}{\mathbf{A}(s,t)}\frac{\partial \mathbf{Q}}{\partial t} + \frac{1}{\mathbf{A}(s,t)}\frac{\partial}{\partial s}(\beta\frac{\mathbf{Q}^2}{\mathbf{A}(s,t)}) + g(S_0 - S_f) + \\ + g\frac{\partial h}{\partial s} - \frac{c_\varpi}{\mathbf{A}(s,t)}\varpi^2 cos\alpha &= \frac{q_t}{\mathbf{A}(s,t)}(v - v_t), \quad \text{on } \Omega_{st}\times(0,t], \end{aligned} \tag{29}$$

where $\mathbf{A}$ is the cross sectional area, $\mathbf{Q}$ is the discharge, $G_s(s,t)$ represents the gain or loss of the stream (i.e. the lateral inflow per unit length of channel), s is the arc-length along the channel, $v = \mathbf{Q}/\mathbf{A}$ the average velocity in s-direction, v_t the velocity component in s-direction of lateral flow from tributaries, the Boussinesq coefficient $\beta = \frac{1}{v^2\mathbf{A}}\int u^2 dA$ (u the flow velocity at a point) and α the wind direction measured from a positive line tangent to s in flow direction. The bottom shear stresses are approximated by using the Chèzy or Manning equations,

$$\begin{aligned} S_f &= \frac{v^2}{C_h^2}\frac{P(h)}{\mathbf{A}(h)}, \quad \text{Chèzy model.} \\ S_f &= \left(\frac{n}{a}\right)^2 v^2 \frac{P^{4/3}(h)}{\mathbf{A}^{4/3}(h)}, \quad \text{Manning model.} \end{aligned} \tag{30}$$

where P is the wetted perimeter of the channel and a is a conversion factor ($a = 1$ for metric units).

6.10.3 Kinematic Wave Model.

When friction and gravity effects dominate over inertia and pressure forces, and, if we neglect the stress due to wind blowing and the coriolis term, the momentum equation becomes

$$S = S_f, \tag{31}$$

and eq. (29)

$$\frac{\partial \mathbf{A}(h)}{\partial t} + \frac{\partial \mathbf{Q}(\mathbf{A}(h))}{\partial s} = G_s, \qquad \text{on } \Omega_{st} \times (0, t], \tag{32}$$

where $\mathbf{A}$ depends, through the geometry of the channel, on the channel water height h. The flow rate $\mathbf{Q}$ under the KWM model is only a function of $\mathbf{A}$ through the friction law.

$$\mathbf{Q} = \gamma \mathbf{A}^m, \tag{33}$$

where $\gamma = C_h\, S^{1/2}\, P^{-1}$ and $m = 3/2$ for the Chèzy friction model, and $\gamma = \bar{a}\, n^{-1}\, S^{1/2}\, P^{-2/3}$ and $m = 5/3$ for the Manning model; $S = (\mathrm{d}h_b/\mathrm{d}s)$ is the slope of the stream bottom.

6.11 Boundary Conditions.

6.11.1 Boundary Conditions to simulate River-Aquifer Interactions/Coupling Term.

The stream/aquifer interaction process occurs between a stream and its adjacent floodplain aquifer. The coupling term is not explicitly included in eq. (22) but it is treated as a boundary flux integral. At a nodal point we can write the coupling,

$$G_s = P/R_f(\phi - h_b - h), \tag{34}$$

where G_s represents the gain or loss of the stream, and the main component is the loss to the aquifer and R_f is the resistivity factor per unit arc length of the perimeter. The corresponding gain to the aquifer is

$$G_a = -G_s\, \delta_{\Gamma_s}, \tag{35}$$

where Γ_s represents the planar curve of the stream and δ_{Γ_s} is a Dirac's delta distribution with a unit intensity per unit length, i.e.

$$\int f(\bar{x})\, \delta_{\Gamma_s}\, \mathrm{d}\Sigma = \int_0^L f(\bar{x}(s))\, \mathrm{d}s. \tag{36}$$

The *stream loss* element set represents this loss, and a typical discretization is shown in fig. 4. The stream loss element is connected to two nodes on the stream and two on the aquifer. If the stream level is over the freatic aquifer level ($h_b + h > \phi$) then the stream losses water to the aquifer and vice versa. Contrary to standard approaches, the coupling term is incorporated through a boundary flux integral that arises naturally in the weak form of the governing equations rather than through a source term.

6.11.2 Initial Conditions. First, Second and Third Kind Boundary Conditions/Absorbent Boundary Condition.

Groundwater flow. In the previous section, the equation that governs subsurface flow was established. In order to obtain a well posed PDE problem, initial and boundary conditions must be superimposed on the flow domain and on its limits. The initial condition for the groundwater problem is a constant hydraulic head in the whole region that obeys levels observed in the basin history.
Now, consider a simply connected region Ω bounded by a closed curve $\partial\Omega$ such that

$\partial\Omega_\phi \cup \partial\Omega_\sigma \cup \partial\Omega_{\phi\sigma} = \partial\Omega$. If the stream is partially penetrating and connected, in a Hydraulic sense, to the aquifer, we set

$$\begin{aligned} \phi &= \phi_0, \quad \text{on } \partial\Omega_\phi \times (0,t] \\ K(\phi-\eta)\frac{\partial\phi}{\partial n} &= \sigma_0, \quad \text{on } \partial\Omega_\sigma \times (0,t] \\ K(\phi-\eta)\frac{\partial\phi}{\partial n} &= C(\phi-h), \quad \text{on } \partial\Omega_{\phi\sigma} \times (0,t] \end{aligned} \tag{37}$$

where ϕ_0 is a given water head, σ_0 is a given flux normal to the flux boundary $\partial\Omega_\sigma$ and C the conductance at the river/stream interface. If a fully penetrating stream is considered,

$$K(\phi-\eta)\frac{\partial\phi}{\partial n} = C(\phi-h), \quad \text{on } \partial\Omega_{\phi\sigma} \times (0,t] \tag{38}$$

Finally, for a perched stream,

$$K(\phi-\eta)\frac{\partial\phi}{\partial n} = C(h_b-h), \quad \text{on } \partial\Omega_{\phi\sigma} \times (0,t] \tag{39}$$

Surface Flow - Fluid Boundary. We recall that the type of a flow in a stream or in an open channel depends on the value of the Froud number $F_r = |\mathbf{u}|/c$ (where $c = \sqrt{gh}$ is the wave celerity), a flow is said

- fluvial, for $|\mathbf{u}| < c$.
- torrential, for $|\mathbf{u}| > c$

Saint-Venant equations. Considering a *Cauchy* problem for the time-like variable $x^{dim+1} = t$ where the solution is given in the subspace $x^{dim+1} = t = 0$ as $\mathbf{U} = \mathbf{U}(\mathbf{x}, t = 0)$ and is determined at subsequent values of t. If the subspace $t = 0$ is bounded by a surface $\partial\Gamma(\mathbf{x})$ then additional conditions have to be imposed on that surface at all values of t. This defines an *initial boundary value problem*. A solution for the system of the first-order PDE's can be written as a superposition of wave-like solutions of the type corresponding to the n-eigenvalues of the matrix $\mathbf{A}^k \cdot \mathbf{n} = \frac{\partial \mathbf{F}_i(\mathbf{U})}{\partial \mathbf{U}}^k \cdot \mathbf{n}, \quad k = 1, .., dim$, being $\mathbf{n}$ the outward unit normal to the boundary edge:

$$\mathbf{U} = \sum_{\alpha=1}^{n} \bar{\mathbf{U}}_\alpha e^{\mathbf{I}(\mathbf{x}\cdot\mathbf{n}-\omega_\alpha t)}, \quad \text{on } \Omega_{st} \times \Gamma_{st} \times (0,t] \tag{40}$$

where summation extends over all eigenvalues λ_α.
As the problem is hyperbolic, n initial conditions for the *Cauchy* problem have to be given to determine the solution. That is, equal number of conditions as unknowns must be imposed at $t = 0$. For initial boundary value problem the n boundary conditions have to be distributed along the limits at all values of t, according to the direction of the propagation of the corresponding waves. If the wave phase velocity, the α-eigenvalue of $\mathbf{A}^k \cdot \mathbf{n}_k$ (i.e. k-wave projected in the interior normal direction $\mathbf{n}$), is positive, the information is propagated inside the domain. Hence, the number of conditions to be imposed for the initial boundary value problem at a given point of $\partial\Gamma$ is equal to the

number of positive eigenvalues of $\mathbf{A}^k \cdot \mathbf{n}_k$ at that point. The total number of conditions remains equal to the total number of eigenvalues (i.e. the order of the system). For the treatment of the boundary conditions we will use the one dimensional projected system and consider the sign of the eigenvalues of $\mathbf{A}^k$ ($u_n + c$ and $u_n - c$). We remark that if $\mathbf{n}$ is the outward unit normal to the boundary edge, an inflow boundary corresponds to $\mathbf{u} \cdot \mathbf{n} < 0$ and an outflow one to $\mathbf{u} \cdot \mathbf{n} > 0$.

Fluvial Boundary.

- inflow boundary: $\mathbf{u}$ specified and the depth h is extrapolated from interior points, or vice versa.
- outflow boundary: depth h specified and velocity field extrapolated from interior points, or vice versa.

Torrential Boundary.

- inflow boundary: $\mathbf{u}$ and the depth h are specified.
- outflow boundary: all variables are extrapolated from interior points.

Solid Wall Condition. We prescribe the simple slip condition over Γ_{slip} ($\subset \Gamma_{st}$)

$$\mathbf{u} \cdot \mathbf{n} = 0 \tag{41}$$

Kinematic Wave Model. The applicability of the kinematic wave as an approximation to dynamic wave was discussed in *Rodríguez*(1995) and, according to Lighthill and Whitham, subcritical flow conditions favor the kinematic approach.
Since one characteristic is propagated inside the domain, only we can specify the water head, the channel section or the discharge at inflow boundaries (see eq. (33)).

6.12 Absorbing boundary conditions

Absorbing boundary conditions are a very useful feature for the solution of advective diffusion problems. They allow the user to put artificial boundaries closer to the interest region, and also accelerate convergence to steady solutions, since provide the highest rate of energy dissipation through the boundaries.

In PETSc-FEM, once you write the flux function for a particular advective-diffusive problem you get absorbing boundary conditions with none or little extra work. There are basically two types of absorbing boundary conditions

- Linear, based on the Jacobian of the flux function, assuming small perturbations about a reference value.
- Based on Riemann invariants (require the writer of the flux function to provide the Riemann invariants for the flux function). (Needs the user to write the Riemann invariants and

6.12.1 Linear absorbing boundary conditions

Starting with the conservation form of an advective system (2), and assuming small perturbations about a mean fluid state $\mathbf{U}(\mathbf{x},t) = \mathbf{U}_0 + \mathbf{U}'(\mathbf{x},t)$, and no source term, then we obtain the linearized form

$$\frac{\partial \mathbf{U}'}{\partial t} + \mathbf{A}_0 \frac{\partial \mathbf{U}'}{\partial x} = 0. \tag{42}$$

where we assume further, that the flow only depends on x the direction normal to the boundary. Let $\mathbf{S}$ be the matrix of right eigenvectors of $\mathbf{A}_0$ so that

$$\mathbf{A}_0 \mathbf{S} = \mathbf{S}\Lambda \tag{43}$$

where $\mathbf{\Lambda} = \text{diag}\{\lambda_1, ..., \lambda_{n_{\text{dof}}}\}$ are the eigenvalues of A_0. Assuming that the system is *"hyperbolic"*, then such a diagonal decomposition is possible, for any state $\mathbf{U}_0$, with real eigenvalues and a non singular matrix $\mathbf{S}$. Multiplying (42) at left by $\mathbf{S}^{-1}$ and defining $V = \mathbf{S}^{-1}\mathbf{U}'$ we obtain a decoupled system of equations

$$\frac{\partial \mathbf{V}}{\partial t} + \mathbf{\Lambda} \frac{\partial \mathbf{V}}{\partial x} = 0. \tag{44}$$

Now, the equation for each *"characteristic component"* v_j of $\mathbf{V}$ is a simple linear transport equation with constant transport velocity λ_j

$$\frac{\partial v_j}{\partial t} + \lambda_j \frac{\partial v_j}{\partial x} = 0. \tag{45}$$

so that the absorbing boundary condition is almost evident. Assuming that we want to solve the equation on the semiplane $x \geq$, so that $x = 0$ is a boundary, then the corresponding absorbing boundary condition is

$$\begin{cases} v_j(0) = 0; & \text{if } \lambda_j \geq 0 \text{ (ingoing boundary)} \\ v_j \text{extrapolated from interior}; & \text{otherwise, (outgoing boundary)} \end{cases} \tag{46}$$

This can be summarized as

$$\mathbf{\Pi}_V^+ \mathbf{V}_0 = 0 \tag{47}$$

where

$$\mathbf{\Pi}_V^+ = \text{diag}\{(1 + \text{sign}(\lambda_j))/2\} \tag{48}$$

is the projection matrix onto the space of incoming waves. As the $\mathbf{\Pi}_V^+$ is a diagonal matrix, with diagonal elements 1 or 0, it trivially satisfies the projection condition $\mathbf{\Pi}_V^+ \mathbf{\Pi}_V^+ = \mathbf{\Pi}_V^+$.

Coming back to the $\mathbf{U}$ basis, we obtain the following first-order, linear absorbing boundary condition

$$\mathbf{\Pi}_U^+(\mathbf{U}_0)\,(\mathbf{U}(0) - \mathbf{U}_0) = 0, \tag{49}$$

where

$$\mathbf{\Pi}_U^\pm = \mathbf{S}\,\mathbf{\Pi}_V^\pm \mathbf{S}^{-1} \tag{50}$$

This condition is perfectly absorbing for small amplitude waves around the state $\mathbf{U}_0$. The main problem with it is that, as the limit state at the boundary $\mathbf{U}_\infty$ is not known a priori, we have to use some *a priori* chosen state $\mathbf{U}_0 \neq \mathbf{U}_\infty$ and then, the projection matrix used $\mathbf{\Pi}_U^+(\mathbf{U}^0)$ will not be equal to $\mathbf{\Pi}_U^+(\mathbf{U}_\infty)$, and then not fully absorbing. We call $\mathbf{U}_0$ the reference state for the absorbing boundary condition. It can even happen that the eigenvalues for the actual state at the boundary change sign with respect to the reference state.

6.12.2 Riemann based absorbing boundary conditions

Let n^+ (n^-) be the number of incoming (outgoing) waves, i.e. the number of positive (negative) eigenvalues of A_0, and assume that the eigenvalues are decreasingly ordered, i,e, $\lambda_j \geq \lambda_k$, if $j < k$. So that the positive eigenvalues are the first n^+ ones. The boundary conditions for the incoming waves (46) can be written as

$$\mathbf{l}_j.(\mathbf{U} - \mathbf{U}_0) = 0, \quad j = 1, \ldots, n^+ \tag{51}$$

where $\mathbf{l}_j$ is a row of $\mathbf{S}^{-1}$, i.e. a *"left eigenvalue"* of $\mathbf{A}_0$. If $\mathbf{U}$ is close to $\mathbf{U}_0$ we can write (51) as

$$\mathbf{l}_j(\mathbf{U}).\,\mathrm{d}\mathbf{U} = 0, \quad j = 1, \ldots, n^+. \tag{52}$$

If this differential forms were *"exact differentials"*, i.e. if

$$\mathbf{l}_j(\mathbf{U}).\,\mathrm{d}\mathbf{U} = \mathrm{d}w_j(\mathbf{U}), \text{ for all } \quad j = 1, \ldots, n_{\mathrm{dof}}, \tag{53}$$

for some functions $w_j(\mathbf{U})$, then we could impose as absorbing boundary conditions

$$w_j(\mathbf{U}) = w_j(\mathbf{U}_0), \quad j = 1, \ldots, n^+. \tag{54}$$

Let's call to these w_j functions *"invariants"*. As there are n_{dof} invariants, we can define as a new representation of the internal state the $\mathbf{w}$ variables. Note that, as $(\partial\mathbf{w}/\partial\mathbf{U}) = S^{-1}$, and $\mathbf{S}$ is non-singular, due to the hyperbolicity of the system, the correspondence between $\mathbf{w}$ and $\mathbf{U}$ is one-to-one.

Assume that the value at the boundary reaches a steady limit value of $\mathbf{U}_\infty$, i.e.

$$\mathbf{U}(0,t) \to \mathbf{U}_\infty, \text{ for } \quad t \to \infty \tag{55}$$

If all the waves were incoming ($n^+ = n_{\mathrm{dof}}$), then the set of boundary conditions (54) would be a set of n_{dof} non-linear equations on the value $\mathbf{U}_\infty$. As the correspondence $\mathbf{U} \to \mathbf{w}$ is one to one, the boundary conditions would mean $\mathbf{w}(\mathbf{U}_\infty) = \mathbf{w}(\mathbf{U}_0)$, and then $\mathbf{U}_\infty = \mathbf{U}_0$. But if the number of incoming waves is $n^+ < n_{\mathrm{dof}}$, then it could happen that $\mathbf{U}_\infty \neq \mathbf{U}_0$. In fact, for a given U_0, the limit value $\mathbf{U}_\infty$ would belong to a curved n^--dimensional curvilinear manifold. Even if the limit state $\mathbf{U}_\infty$=$\mathbf{U}_0$, we can proved to be perfectly absorbing, since, as $\mathbf{U} \to \mathbf{U}_\infty$ at the boundary, we can expand each of the conditions around $\mathbf{U}_\infty$ and it would result in an equation similar to (52) but centered about $\mathbf{U}_\infty$,

$$\mathbf{l}_j(\mathbf{U}_\infty).(\mathbf{U} - \mathbf{U}_\infty) = 0, \quad j = 1, \ldots, n^+. \tag{56}$$

The problem is that in general the differentials are not exact. *"Riemann invariants"* are functions that satisfy (53) under some restrictions on the flow. For instance, Riemann invariants can be computed for compressible gas flow if we assume that the flow is isentropic. They are

$$\begin{aligned}
w_1 &= s = \log(p/\rho^\gamma), & \lambda_1 &= u, & &\text{(entropy)};\\
w_{2,3} &= u \pm \frac{2a}{\gamma - 1}, & \lambda_{2,3} &= u \pm a, & &\text{(acoustic waves)};\\
w_{4,5} &= \mathbf{u} \cdot \hat{\mathbf{t}}_{1,2}; & \lambda_{4,5} &= u, & &\text{(vorticity waves)}.
\end{aligned} \tag{57}$$

Boundary conditions based on Riemann invariants are, then, absorbing in some particular cases.

6.12.3 Absorbing boundary conditions based on last state

Another possibility is to linearize around the last state $\mathbf{U}^n$, i.e.

$$\mathbf{\Pi}_U^+(\mathbf{U}(0,t^n))\,(\mathbf{U}(0,t^{n+1}) - \mathbf{U}(0,t^n)) = 0. \tag{58}$$

This equation is always perfectly absorbing in the limit, because we are always linearizing about a state that, in the limit, will tend to $\mathbf{U}_\infty$ and doesn't need the computation of Riemann invariants (which could be unknown for certain problems). Also, this boundary condition is fully absorbing even in the case of inversion of the sense of propagation of waves.

The drawback is that we have no external control on the internal states, i.e. the limit internal state does not depend on some external value (as the $\mathbf{U}_0$ used for the Riemann based absorbing boundary conditions), but on the internal state. That means, for instance, that if the internal computational scheme tends to produce some error (due to non conservativity, or rounding errors), the internal state would drift constantly.

A good compromise may be to use Riemann based (54) or linear absorbing boundary conditions (49) at inlet and absorbing boundary conditions based on last state (58) at outlet. As the error tends to propagate more intensely towards the outlet boundary, it is preferably to use strongly absorbing boundary conditions there, whereas the linearly absorbing or Riemann invariant boundary conditions upstream avoid the drift.

6.12.4 Finite element setup

Assume that the problem is 1D, with a constant mesh size h and time step Δt, so that nodes are located at positions $x_k = kh$, $k = 0, \ldots, \infty$. Let $\mathbf{U}_j^n$ be the state at node j, time step n. FEM discretisation of the system of equations with *"natural"* boundary conditions leads to a system of the form

$$\begin{aligned}
\mathbf{F}_0(\mathbf{U}_0^{n+1}, \mathbf{U}_1^{n+1}) &= \mathbf{R}_0^{n+1} \\
\mathbf{F}_1(\mathbf{U}_0^{n+1}, \mathbf{U}_1^{n+1}, \mathbf{U}_2^{n+1}) &= \mathbf{R}_1^{n+1} \\
\vdots \quad &= \quad \vdots \\
\mathbf{F}_k(\mathbf{U}_{k-1}^{n+1}, \mathbf{U}_k^{n+1}, \mathbf{U}_{k+1}^{n+1}) &= \mathbf{R}_k^{n+1} \\
\vdots \quad &= \quad \vdots
\end{aligned} \tag{59}$$

where the $\mathbf{F}_k()$ are non-linear functions and the $\mathbf{R}_k$ possibly depends on the previous values $\mathbf{U}_k^n$. *"Imposing boundary conditions"* means to replace some of the equations in the first row ($k = 0$) for other equations. Recall that, in order to balance the number of equations and unknowns, we must specify which of the equations are discarded for each equations that is added. For instance, when imposing conditions on primitive variables it is usual to discard the energy equation if pressure is imposed, to discard the continuity equation if density is imposed and to discard the j-th component of the momentum equation if the j-th component of velocity is imposed. On solid walls, the energy equation is discarded if temperature is imposed. Some of these *"pairings"* equation/unknown are more clear than others.

So that, when imposing absorbing boundary conditions we have not only to specify the new conditions, but also which equations are discarded. Note that this is not necessary if all the variables are imposed at the boundary, for instance in a supersonic outlet. This suggests to generate appropriate values even for the outgoing waves, for instance, by extrapolation from the interior values.

6.12.5 Extrapolation from interiors

For a linear system in characteristic variables (45) we could replace all the first row of equations by

$$v_{j0}^{n+1} = \begin{cases} 0; & j = 1, .., n^+ \\ \sum_{p=0}^{m} c_p v_{jp}^n; & j = n^+ + 1, \dots, n_{\text{dof}} \end{cases}; \tag{60}$$

which can be put in matricial form as

$$\begin{aligned} \mathbf{\Pi}_V^+ \mathbf{V}_0^{n+1} &= 0 \\ \mathbf{\Pi}_V^- (\mathbf{V}_0^{n+1} - \sum_{p=0}^{m} c_p \mathbf{V}_p^n) &= 0 \end{aligned} \tag{61}$$

where the c_p's are appropriate coefficients that provide an extrapolation to the value $\mathbf{v}_0^{n+1}$ from the values at time $\mathbf{v}_0^{n+1}$ Note that these represents $2n_{\text{dof}}$ equations, but as $\mathbf{\Pi}_V^{\pm}$ have rank $n^{\pm}$ there are, in total, n_{dof} linearly independent equations. As the $n^+ + 1 \leq j \leq n_{\text{dof}}$ rows in the first row of equations (corresponding to incoming waves) are null and *vice versa* for the outgoing waves, we can add both blocks of equations to add a single set of n_{dof} equations

$$\mathbf{\Pi}_V^+ \mathbf{V}_0^{n+1} + \mathbf{\Pi}_V^- (\mathbf{V}_0^{n+1} - \sum_{p=0}^{m} c_p \mathbf{V}_p^n) = 0 \tag{62}$$

and, coming back to the $\mathbf{U}$ basis

$$\mathbf{\Pi}_U^+ (\mathbf{U}_0^{n+1} - \mathbf{U}_0) + \mathbf{\Pi}_U^- (\mathbf{U}_0^{n+1} - \sum_{p=0}^{m} c_p \mathbf{U}_p^n) = 0 \tag{63}$$

The modified version of the FEM system of equations (59) that incoroporates the absorbing boundary conditions is, then

$$\begin{aligned} \mathbf{\Pi}_U^+ (\mathbf{U}_0^{n+1} - \mathbf{U}_0) + \mathbf{\Pi}_U^- (\mathbf{U}_0^{n+1} - \sum_{p=0}^{m} c_p \mathbf{U}_p^n) &= 0 \\ \mathbf{F}_1(\mathbf{U}_0^{n+1}, \mathbf{U}_1^{n+1}, \mathbf{U}_2^{n+1}) &= \mathbf{R}_1^{n+1} \\ \vdots \quad &= \quad \vdots \\ \mathbf{F}_k(\mathbf{U}_{k-1}^{n+1}, \mathbf{U}_k^{n+1}, \mathbf{U}_{k+1}^{n+1}) &= \mathbf{R}_k^{n+1} \\ \vdots \quad &= \quad \vdots \end{aligned} \tag{64}$$

6.12.6 Avoiding extrapolation

For linear systems, equation (59) is of the form

$$\begin{aligned} \frac{\mathbf{U}_0^{n+1}-\mathbf{U}_0^n}{\Delta t}+\mathbf{A}\,\frac{\mathbf{U}_1^{n+1}-\mathbf{U}_0^n}{h}&=0;\\ \frac{\mathbf{U}_k^{n+1}-\mathbf{U}_k^n}{\Delta t}+\mathbf{A}\,\frac{\mathbf{U}_{k+1}^{n+1}-\mathbf{U}_{k-1}^n}{2h}&=0,\quad k\geq 1 \end{aligned} \tag{65}$$

We have made a lot of simplifications here, no source or upwind terms, and a simple discretization based on centered finite differences. Alternatively, it can be thought as a pure Galerkin FEM discretization with mass lumping. In the base of the characteristic variables $\mathbf{V}$ this could be written as

$$\begin{aligned} \frac{\mathbf{V}_0^{n+1}-\mathbf{V}_0^n}{\Delta t}+\mathbf{\Lambda}\,\frac{\mathbf{V}_1^{n+1}-\mathbf{V}_0^n}{h}&=0;\\ \frac{\mathbf{V}_k^{n+1}-\mathbf{V}_k^n}{\Delta t}+\mathbf{\Lambda}\,\frac{\mathbf{V}_{k+1}^{n+1}-\mathbf{V}_{k-1}^n}{h}&=0,\quad k\geq 1. \end{aligned} \tag{66}$$

For the linear absorbing boundary conditions (49) we should impose

$$\mathbf{\Pi}_V^+(\mathbf{V}_{\text{ref}})\,(\mathbf{V}_0-\mathbf{V}_{\text{ref}})=0. \tag{67}$$

while discarding the equations corresponding to the incoming waves in the first rows of (66). Here $\mathbf{U}_{\text{ref}}/\mathbf{V}_{\text{ref}}$ is the state about which we make the linearization. This can be done, via Lagrange multipliers in the following way

$$\begin{aligned} \mathbf{\Pi}_V^+(\mathbf{V}_{\text{ref}})\,(\mathbf{V}_0-\mathbf{V}_{\text{ref}})+\mathbf{\Pi}_V^-(\mathbf{V}_{\text{ref}})\,\mathbf{V}_{lm}&=0,\\ \frac{\mathbf{V}_0^{n+1}-\mathbf{V}_0^n}{\Delta t}+\mathbf{\Lambda}\,\frac{\mathbf{V}_1^{n+1}-\mathbf{V}_0^n}{h}+\mathbf{\Pi}_V^+\,\mathbf{V}_{lm}&=0;\\ \frac{\mathbf{V}_k^{n+1}-\mathbf{V}_k^n}{\Delta t}+\mathbf{\Lambda}\,\frac{\mathbf{V}_{k+1}^{n+1}-\mathbf{V}_{k-1}^n}{2h}&=0,\quad k\geq 1. \end{aligned} \tag{68}$$

where $\mathbf{V}_{lm}$ are the Lagrange multipliers for imposing the new conditions. Note that, if j is an incoming wave ($\lambda_j >= 0$), then the equation is of the form

$$\begin{aligned} v_{j0}-v_{\text{ref}0}&=0\\ \frac{v_{j0}^{n+1}-v_{j0}^n}{\Delta t}+\lambda_j\frac{v_{j1}^{n+1}-v_{j0}^n}{h}+v_{j,lm}&=0\\ \frac{v_{jk}^{n+1}-v_{jk}^n}{\Delta t}+\lambda_j\frac{v_{j,k+1}^{n+1}-v_{jk}^n}{2h}&=0,\quad k\geq 1 \end{aligned} \tag{69}$$

Note that, due to the $v_{j,lm}$ Lagrange multiplier, we can solve for the v_{jk} values from the first last rows, the value of the multiplier $v_{j,lm}$ *"adjusts"* itself in order to relax the equations in the second row.

On the other hand, for the outgoing waves ($\lambda_j < 0$), we have

$$\begin{aligned} v_{j,lm}&=0\\ \frac{v_{j0}^{n+1}-v_{j0}^n}{\Delta t}+\lambda_j\frac{v_{j1}^{n+1}-v_{j0}^n}{h}&=0\\ \frac{v_{jk}^{n+1}-v_{jk}^n}{\Delta t}+\lambda_j\frac{v_{j,k+1}^{n+1}-v_{jk}^n}{2h}&=0,\quad k\geq 1 \end{aligned} \tag{70}$$

So that the solution coincides with the unmodified orginal FEM equation, and $v_{j,lm} = 0$.

Coming back to the $\mathbf{U}$ basis, we have

$$\begin{aligned} \boldsymbol{\Pi}_U^+(\mathbf{U}_{\text{ref}})\,(\mathbf{U}_0 - \mathbf{U}_{\text{ref}}) + \boldsymbol{\Pi}_U^-(\mathbf{U}_{\text{ref}})\,\mathbf{U}_{lm} &= 0, \\ \frac{\mathbf{U}_0^{n+1} - \mathbf{U}_0^n}{\Delta t} + \mathbf{A}\,\frac{\mathbf{U}_1^{n+1} - \mathbf{U}_0^n}{h} + \boldsymbol{\Pi}_U^+\,\mathbf{U}_{lm} &= 0; \\ \frac{\mathbf{U}_k^{n+1} - \mathbf{U}_k^n}{\Delta t} + \mathbf{A}\,\frac{\mathbf{U}_{k+1}^{n+1} - \mathbf{U}_{k-1}^n}{2h} &= 0, \quad k \geq 1. \end{aligned} \tag{71}$$

And finally, coming back to the FEM equations (59),

$$\begin{aligned} \boldsymbol{\Pi}_U^+(\mathbf{U}_{\text{ref}})\,(\mathbf{U}_0 - \mathbf{U}_{\text{ref}}) + \boldsymbol{\Pi}_U^-(\mathbf{U}_{\text{ref}})\,\mathbf{U}_{lm} &= 0, \\ \mathbf{F}_0(\mathbf{U}_0^{n+1}, \mathbf{U}_1^{n+1}) + \boldsymbol{\Pi}_U^+\,\mathbf{U}_{lm} &= \mathbf{R}_0^{n+1} \\ \mathbf{F}_1(\mathbf{U}_0^{n+1}, \mathbf{U}_1^{n+1}, \mathbf{U}_2^{n+1}) &= \mathbf{R}_1^{n+1} \\ \vdots \quad &= \quad \vdots \\ \mathbf{F}_k(\mathbf{U}_{k-1}^{n+1}, \mathbf{U}_k^{n+1}, \mathbf{U}_{k+1}^{n+1}) &= \mathbf{R}_k^{n+1} \\ \vdots \quad &= \quad \vdots \end{aligned} \tag{72}$$

In conclusion, in this setup we do not need to make extrapolations to the variables, and then there is no need to have a structured line of nodes near the boundary. It's only required to have an additional fictitious node at the boundary in order to hold the Lagrange multiplier unknowns U_{lm}, and to add the absorbing boundary equation (first row of (72) for these nodes.

6.12.7 Flux functions with enthalpy.

When the flux function has an enthalpy term that is not the identity, then the expressions for the change of basis are somewhat modified, and also the projectors. An advective diffusive-system with a *"generalized enthalpy function"* $\mathcal{H}(\mathbf{U})$ is an extension of the form (2) and can be written as

$$\frac{\partial}{\partial t}\,\mathcal{H}(\mathbf{U}) + \frac{\partial \mathcal{F}_i(U)}{\partial x_i} = 0 \tag{73}$$

The heat equation can be naturally put in this way. Also, the gas dynamics equations for compressible flow can be put in this form if we put the equations in *"conservative form"* but use the *"primitive variables"* $\mathbf{U} = [\rho, \mathbf{u}, p]^T$ as the main main variables for the code. This has the advantage of using a conservative form of the equations and, at the same time, allows an easy imposition of Dirichlet boundary conditions that are normally set in terms of the primitive variables. In this case $\mathbf{U}$ are the primitive variables, and the generalized enthalpy $\mathcal{H}(\mathbf{U})$ is the vector of conservative variables. We call the generalized *"heat content matrix"* $\mathbf{C}_p$ as

$$\mathbf{C}_p = \frac{\partial \mathcal{H}(\mathbf{U})}{\partial \mathbf{U}} \tag{74}$$

and (2) can be put in quasi-linear form as

$$\mathbf{C}_p\,\frac{\partial \mathbf{U}}{\partial t} + \mathbf{A}_i\,\frac{\partial U}{\partial x_i} = 0 \tag{75}$$

Note that this can be brought to the quasi-linear form (42) (i.e., without the $\mathbf{C}_p$) if we multiply the equation at left by $\mathbf{C}_p^{-1}$ and define new flux Jacobians as

$$\tilde{\mathbf{A}}_i = \mathbf{C}_p^{-1}\mathbf{A}_i, \tag{76}$$

So that, basically, the extension to systems with generalized enthalpy is to replace the Jacobians, by the modified Jacobians (76). The modified expression for the projectors is

$$\mathbf{\Pi}_U^{\pm} = \mathbf{C}_p\,\mathbf{S}\,\mathbf{\Pi}_V^{\pm}\mathbf{S}^{-1} \tag{77}$$

6.12.8 Absorbing boundary conditions available

At the moment of writing this, we have three possible combinations of boundary conditions.

Using extrapolation from the interior. These is the elemset `<system>_abso`. The number of nodes per element $n_{\rm el}$ must be not lower than 4. The first $n_{\rm el}-2$ nodes are used for a second order extrapolation of the outgoing wave. The $n_{\rm el}-1$-th node is the node with Lagrange multipliers, and the $n_{\rm el}$-th node is used to set the reference value. For instance, for an absorbing elment of $n_{\rm el}=5$ nodes, we would have 3 internal nodes, and the data would loke like this (see figure 5)

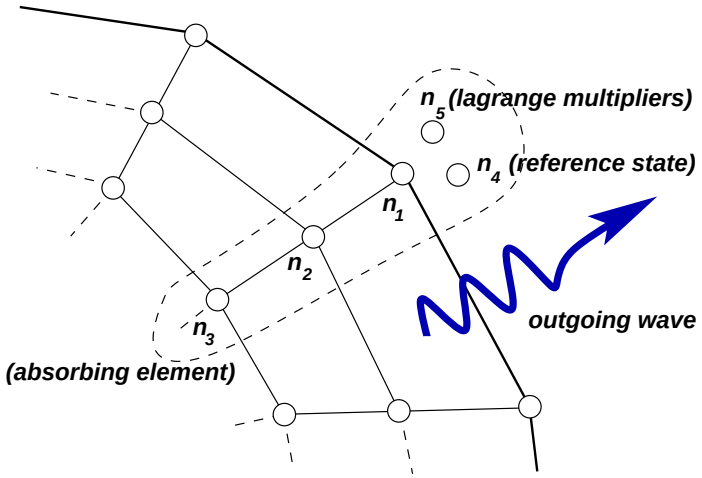


Figure 5: Absorbing element.

```
elemset gasflow_abso 5
normal <nx> <ny>
__END_HASH__
<n1> <n2> <n3> <n4> <n5>
...
__END_ELEMSET__

end_elemsets
```

```
fixa
<n4> 1 <rho_ref>
<n4> 2 <u_ref>
<n4> 3 <v_ref>
<n4> 4 <p_ref>
__END_FIXA__
```

- Each element has 5 nodes, first three are real nodes (i.e. not fictitious) numbered from the boundary to the interior. Fourth node is reserved for Lagrange multipliers, and the fifth node is set to the reference value.
- The `normal` option is used to define the normal to the boundary. It can be set as a constant vector per elemset (usually for plane boundaries, as in the example above), or as a per element value. In this last case we would have something like this
  ```
  elemset gasflow_abso 5
  props normal[2]
  normal <nx> <ny>
  __END_HASH__
  <n1> <n2> <n3> <n4> <n5> <nx> <ny>
  ...
  __END_ELEMSET__
  ```
  The normal need not be entered with a high precision. If the vector entered is not exactly normal to the boundary, then the condition will be absorbing for waves whose *"group velocity vector"* is parallel with this vector.
- Note the the `fixa` section for the values of the reference node. In this case (gas dynamics, elemset `gasflow`) we set the four fields ($n_{\text{dim}} = 2$) to the reference values.
- The reference values can be made time dependent in an explicit way by using a `fixa_amplitude` section instead of a `fixa` section.
- Using this absorbing boundary condition requires the flow to have implemented the `Riemman_Inv()` method. If this is not the case, then the program will stop with a message like
- *For the* `gasflow` *elemset:* If the option `linear_abso` is set to false (0), then the Riemman invariants for gas dynamics are used, and the state reference value is used for computing the reference Riemman invariants. If `linear_abso` is set to true (1), then the linear absorbing boundary conditions are imposed.

Not using extrapolation from the interior. These is the elemset `<system>_abso2`. The number of nodes per element n_{el} is 3. The first node is the node at the boundary. The second node is the node with Lagrange multipliers, and the third node is used to set the reference value. The data would look like this (see figure 6)

```
elemset gasflow_abso2 3
normal <nx> <ny>
__END_HASH__
<n1> <n2> <n3>
...
__END_ELEMSET__
```

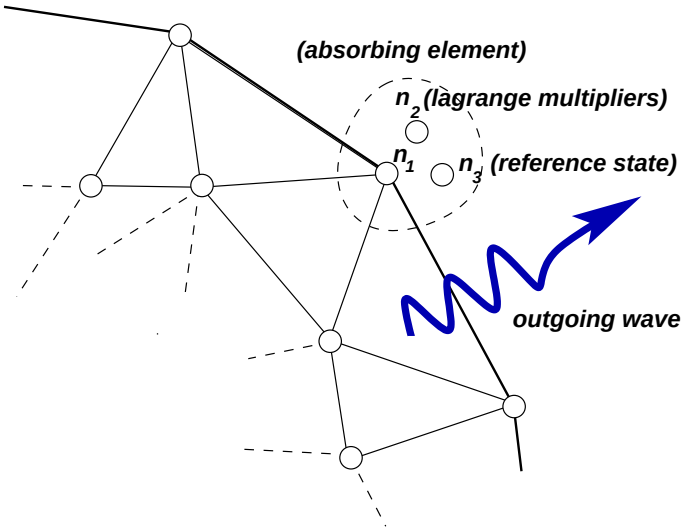


Figure 6: Absorbing element.

```
end_elemsets

fixa
<n4> 1 <rho_ref>
<n4> 2 <u_ref>
<n4> 3 <v_ref>
<n4> 4 <p_ref>
__END_FIXA__
```

- As before, the `normal` property is used for computing the direction normal to the boundary.
- If the `use_old_state_as_ref` flag is set to true (1), then the reference state is taken as the state o the state of the boundary node at the previous time step. In this case the state of the third node is ignored. On the other hand if it is set to false (0), then the state of the third node is used as the reference state.
- This type of boundary condition doesn't need the implementation of the Riemman invariants methd, but it needs the methods `get_Cp()` and `get_Ajac()`.

7 The Navier-Stokes module

7.1 LES implementation

The Smagorisky LES model for the Navier-Stokes module follows ... The implementation under PETSc-FEM has presents the following particularities

- Wall boundary conditions are implemented as *"mixed type"*.
- The van Driest damping factor introduces non-localities in the sense that the turbulent viscosity at a volume element depends on the state of the fluid at a wall.

7.1.1 The wall elemset

Wall boundary conditions have been implemented via a `wall` elemset. This is a surface element that computes, given the velocities at the nodes, the tractions corresponding to this velocities, for a given law of wall. Also, this shear velocities as stored internally in the element, so that the volume elements can get them and compute the van Driest damping factor. This requires to find, for each volume element, the nearest wall element. This is done *before* the time loop, with the ANN (Approximate Nearest Neighbor) library.

7.1.2 The mixed type boundary condition

The contribution to the momentum equations from the wall element is

$$R_{ip} = \int_{\Sigma_e} t_p N_i \, \mathrm{d}\Sigma \tag{78}$$

where R_{ip} is the contribution to the residual of the p-th momentum equation of the node i. N_i is the shape function of node i and t_p are the tractions on the surface of the element Σ_e. The wall law is in general of the form

$$\frac{u}{u^*} = f(y^+) \tag{79}$$

where u is the tangent velocity, u^* the shear velocity $u^* = \sqrt{\tau_w/\rho}$, and $y^+ = yu^*/\nu$, the non-dimensional distance to the wall. We have several possibilities regarding the positioning of the computational boundary. We first discuss the simplest, that is to set the computational boundary at a fixed y^+ position. Note, that this means that the real position od the boundary y^+ changes during iteration. In this case (79) can be rewritten as

$$\tau_w = g(u)\, u \tag{80}$$

where

$$\tau_w = g(u)\, u = \rho \left(\frac{u}{f(y^+)} \right)^2 \tag{81}$$

or

$$g(u) = \frac{\rho}{f(y^+)^2} u \tag{82}$$

The traction on the wall element is assumed to be parallel to the wall and in opposite direction to the velocity vector, that is

$$t_p = -g(u) u_p \tag{83}$$

Replacing in (78) the residual term is

$$R_{ip} = -\int_{\Sigma_e} g(u) u_p N_i \, \mathrm{d}\Sigma \tag{84}$$

The Jacobian of the residual with respect to the state variables, needed for the Newton-Raphson algorithm is

$$\begin{aligned} J_{ip,jq} = -\frac{\partial R_{ip}}{\partial u_{jq}} = \int_{\Sigma_e} \frac{\partial}{\partial u_{jq}} (g(u) u_p)\, N_i \, \mathrm{d}\Sigma \\ \int_{\Sigma_e} \left(g(u) \frac{\partial u_p}{\partial u_{jq}} + g'(u)\, u_p \frac{\partial u}{\partial u_{jq}} \right) N_i \, \mathrm{d}\Sigma \end{aligned} \tag{85}$$

but

$$u_p = \sum_l u_{lp} N_l \tag{86}$$

so that

$$\begin{aligned} \frac{\partial u_p}{\partial u_{jq}} &= \sum_l \frac{\partial u_{lp}}{\partial u_{jq}} N_l \\ &= \sum_l \delta_{lj}\, \delta_{pq} N_l \\ &= \delta_{pq}\, N_j \end{aligned} \tag{87}$$

Similarly,

$$u^2 = u_p\, u_p = u_{lp} N_l\, u_{mp} N_m \tag{88}$$

and

$$2u\, \frac{\partial u}{\partial u_{jq}} = 2u_p \frac{\partial u_p}{\partial u_{jq}} \tag{89}$$

so that

$$\frac{\partial u}{\partial u_{jq}} = \frac{u_p}{u} N_j \tag{90}$$

Replacing in (85),

$$J_{ip,jq} = \int_{\Sigma_e} \left(g(u)\, \delta_{pq} + \frac{g'(u)}{u}\, u_p\, u_q \right) N_i N_j \, \mathrm{d}\Sigma \tag{91}$$

7.1.3 The van Driest damping factor. Programming notes

This is a non-standard issue, since the computation of one volume element requires information of other (wall) elements. First we compute the wall element that is associated to each volume element. `assemble()` is called with `jobinfo="build_nneighbor_tree"`. This jobinfo is acknowledged only by the wall elemsets which compute their geometrical center and put them in the `data_pts` STL array. Then, this is passed to the ANN package which computes the octree. All this is cached in the constructor of a `WallData` class. After this a call to `assemble()` with `jobinfo="get_nearest_wall_element"` is acknowledged by all the volume elemsets, that compute for each volume element the neares wall element. This is stored as an *"integer per element property"* in the volume elemsets. In order to reduce memory requirements only an index in the `data_pts` array is stored. As several wall elemsets may exists, an array of `pair<int,elemset *>` is used to store pointers to the `data_pts` array in order to know to which wall elemset the given index in `data_pts` belongs.

```
vector<double> *data_pts_ = new vector<double>;
vector<ElemToPtr> *elemset_pointer = new vector<ElemToPtr>;
WallData *wall_data;
if (LES) {

  VOID_IT(argl);
```

```
  argl.arg_add(data_pts_,USER_DATA);
  argl.arg_add(elemset_pointer,USER_DATA);
  Elemset *elemset=NULL;
  argl.arg_add(elemset,USER_DATA);
  ierr = assemble(mesh,argl,dofmap,
                  "build_nneighbor_tree",&time); CHKERRA(ierr);
  PetscPrintf(PETSC_COMM_WORLD,"After nearest neighbor tree.\n");

  wall_data = new WallData(data_pts_,elemset_pointer,ndim);

  //---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
  // Find nearest neighbor for each volume element
  VOID_IT(argl);
  argl.arg_add(wall_data,USER_DATA);
  ierr = assemble(mesh,argl,dofmap,"get_nearest_wall_element",
           &time); CHKERRA(ierr);
}
```

In the `jobinfo="build_nneighbor_tree"` call to `assemble()` a loop over all the elements in the elemset, ignoring to what processor it belongs, must be performed. Otherwise, each processor loads in `data_pts` only the coordinates of the elements that belong to him. A possible solution is, after the loop, to exchange the information among the processors, but the simplest solution id to simply bypass the element selection with `compute_this_elem()` with a call like

```
  for (int k=el_start; k<=el_last; k++) {
    if (!build_nneighbor_tree ||
          comp_shear_vel ||
          compute_this_elem(k,this,myrank,iter_mode))) continue;
...
```

That means that for `jobinfo="build_nneighbor_tree"` and `"comp_shear_vel"` the normal element selection is bypassed.

7.2 Options

General options:

- `int A_van_Driest` (default=0):
  If `A_van_Driest=0` then the van Driest damping factor is not used (found in file: `ns.cpp`)
- `int activate_debug` (default=0):
  Activate debugging (found in file: `ns.cpp`)
- `int activate_debug_memory_usage` (default=0):
  Activate report of memory usage (found in file: `ns.cpp`)
- `int activate_debug_print` (default=0):
  Activate printing in debugging (found in file: `ns.cpp`)

- `double alpha` (default=`1.`):
  Trapezoidal method parameter. `alpha=1` : Backward Euler. `alpha=0` : Forward Euler. `alpha=0.5` : Crank-Nicholson. (found in file: `ns.cpp`)
- `double displ_factor` (default=`0.1`):
  Scales displacement for ALE-like mesh relocation. (found in file: `ns.cpp`)
- `double Dt` (default=`0.`):
  The time step. (found in file: `ns.cpp`)
- `int fractional_step` (default=`0`):
  Use fractional step or TET algorithm (found in file: `ns.cpp`)
- `string fractional_step_solver_combo` (default=`iisd`):
  Solver combination for the fractional step method. May be `iisd`, `lu`, `global_gmres`. (found in file: `ns.cpp`)
- `int fractional_step_use_petsc_symm` (default=`1`):
  Fractional step uses symmetric matrices (only CG iterative KSP). (found in file: `ns.cpp`)
- `string gather_file` (default=`gather.out`):
  Print values in this file (found in file: `ns.cpp`)
- `int LES` (default=`0`):
  Use the LES/Smagorinsky turbulence model. (found in file: `ns.cpp`)
- `int measure_performance` (default=`0`):
  Measure performance of the 'comp_mat_res' jobinfo. (found in file: `ns.cpp`)
- `int ndim` (default=`3`):
  Dimension of the problem. (found in file: `ns.cpp`)
- `vector<double> newton_relaxation_factor` (default= `(none)`):
  Relaxation parameter for Newton iteration. Several values may be entered in the form

  ```
  newton_relaxation_factor w1 n1 w2 n2 .... wn
  ```

  that means: Take relaxation factor `w1` for the first `n1` steps, `w2` for the following `n2` steps and so on until `w_{n-1}`. `wn` is taken for all subsequent steps. Normally one takes a conservative (said 0.5) relaxation factor for the first steps and then let full Newton (i.e. `w=1`) for the rest. For instance, the line

  ```
  newton_relaxation_factor 0.5 3 1.
  ```

  means: take $w = 0.5$ for the first 3 steps, and then use $w = 1$. (found in file: `ns.cpp`)
- `int nfile` (default=`1`):
  Sets the number of files in the "rotary save" mechanism. (see 7.2) (found in file: `ns.cpp`)
- `int ngather` (default=`0`):
  Number of "gathered" quantities. (found in file: `ns.cpp`)

- `int nnwt` (default=1):
  Number of inner iterations for the global non-linear Newton problem. (found in file: `ns.cpp`)
- `int nrec` (default=1000000):
  Sets the number of states saved in a given file in the "rotary save" mechanism (see 7.2 (found in file: `ns.cpp`)
- `int nsave` (default=10):
  Sets the save frequency in iterations (found in file: `ns.cpp`)
- `int nsaverot` (default=100):
  Sets the frequency save for the "rotary save" mechanism. Sometimes it is interesting to save the state vector with a certain frequency in a "append" manner, i.e. appending the state vector at the end of the file. However, this posses the danger of storing too much amount of data if the user performs a very long run. The "rotary save" mechanism allows writing only a certain amount of the recent states. The mechanism basically saves the state vector each `nsaverot` steps appending to the a file. The name of the file is contructed from a pattern set by the user via the `save_file_pattern` entry, by replacing `%d` by `0` "à la" `printf()`. For instance, if `save_file_pattern` i set to `file%d.out` then the state vectors are appended to `file0.out`. When the number of written states reach the `nrec` count, the file is reset to 0, and the saving continues from the start of the file. However, if `nfile` is greater than one, then the state vector are continued to be stored in file `file1.out` and so on. When the number of files `nfile` is reached, the saving continues in file '0'.
  More precisely, the saving mechanism is described by the following pseudo-code:

```
Read state vector from 'initial_state' file into x^0, n=0;
for (i=0; i<nstep; i++) {
   advance x^n to x^{n+1};
   if (n % nsaverot == 0) {
           j <- n/nsaverot;
           k <- j % nrec;
           l <- j / nrec;
           if (k==0) { rewind file l; }
           append state vector to file l;
   }
}
```

  (found in file: `ns.cpp`)
- `int nsome` (default=10000):
  Sets the save frequency in iterations for the "print some" mechanism. The "print some" mechanism allows the user to store the variables of some set of nodes with some frequency. The nodes are entered in a separate file whose name is given by a `print_some_file` entry in the general options, one node per line. The entry `nsome` indicates the frequency (in steps) at which the data

is saved and `save_file_some` the name of the file to save in. (found in file: `ns.cpp`)

- `int nstep` (default=`10000`):
  The number of time steps. (found in file: `ns.cpp`)
- `int print_linear_system_and_stop` (default=`0`):
  After computing the linear system solves it and prints Jacobian, right hand side and solution vector, and stops. (found in file: `ns.cpp`)
- `int print_residual` (default=`0`):
  Print the residual each `nsave` steps. (found in file: `ns.cpp`)
- `string print_some_file` (default=`<none>`):
  Name of file where to read the nodes for the "print some" feature. (found in file: `ns.cpp`)
- `int report_option_access` (default=`1`):
  Print, after execution, a report of the times a given option was accessed. Useful for detecting if an option was used or not. (found in file: `ns.cpp`)
- `int reuse_mat` (default=`0`):
  Use fractional step or TET algorithm (found in file: `ns.cpp`)
- `string save_file` (default=`outvector.out`):
  The name of the file to save the state vector. (found in file: `ns.cpp`)
- `string save_file_pattern` (default=`outvector%d.out`):
  The pattern to generate the file name to save in for the rotary save mechanism. (found in file: `ns.cpp`)
- `string save_file_some` (default=`outvsome.out`):
  Name of file where to save node values for the "print some" feature. (found in file: `ns.cpp`)
- `int save_file_some_append` (default=`1`):
  Access mode to the "some" file. If 0 rewind file. If 1 append to previous results. (found in file: `ns.cpp`)
- `int solve_system` (default=`1`):
  Solve system before `print_linear_system_and_stop` (found in file: `ns.cpp`)
- `string solver` (default=`petsc`):
  Type of solver. May be `iisd` or `petsc` . (found in file: `ns.cpp`)
- `string solver_mom` (default=`petsc`):
  Type of solver for the projection and momentum steps (fractional-step). May be `iisd` or `petsc` . (found in file: `ns.cpp`)
- `double start_comp_time` (default=`0.`):
  Time to start computations (found in file: `ns.cpp`)
- `string stdout_file` (default=`none`):
  If set, redirect output to this file. (found in file: `ns.cpp`)

- `int steady` (default=0):
  Flag if steady solution or not (uses Dt=inf). If `steady` is set to 1, then the computations are as if $\Delta t = \infty$. The value of `Dt` is used for printing etc... If `Dt` is not set and `steady` is set then `Dt` is set to one. (found in file: `ns.cpp`)
- `int stop_mom` (default=0):
  After computing the linear system for the predictor/momentum step print right hand side and solution vector, and stops. (found in file: `ns.cpp`)
- `int stop_on_step` (default=1):
  After computing the linear system for the predictor/momentum step print right hand side and solution vector, and stops. (found in file: `ns.cpp`)
- `int stop_poi` (default=0):
  After computing the linear system for the poisson step print right hand side and solution vector, and stops. (found in file: `ns.cpp`)
- `int stop_prj` (default=0):
  After computing the linear system for the projection step print right hand side and solution vector, and stops. (found in file: `ns.cpp`)
- `double tol_newton` (default=1e-8):
  Tolerance to solve the non-linear system (global Newton). (found in file: `ns.cpp`)
- `int update_jacobian_iters` (default=1):
  Update jacobian only until n-th Newton subiteration. Don't update if null. (found in file: `ns.cpp`)
- `int update_jacobian_start_iters` (default=INF):
  Update jacobian each n-th Newton iteration (found in file: `ns.cpp`)
- `int update_jacobian_start_steps` (default=INF):
  Update jacobian each n-th time step. (found in file: `ns.cpp`)
- `int update_jacobian_steps` (default=0):
  Update jacobian each n-th time step. (found in file: `ns.cpp`)
- `int update_wall_data` (default=0):
  If 0: compute `wall_data` info only once. Otherwise refresh each `update_wall_data` steps. (found in file: `ns.cpp`)
- `int use_iisd` (default=0):
  Use IISD (Interface Iterative Subdomain Direct) or not. (found in file: `ns.cpp`)
- `int verify_jacobian_with_numerical_one` (default=0):
  Computes jacobian of residuals and prints to a file. May serve to debug computation of the analytic jacobians. (found in file: `ns.cpp`)

Elemset "`nsitetlesfm2`":

- `int ALE_flag` (default=0):
  Flag to **turn on** ALE computation (found in file: `nsitetlesfm2.cpp`)

- `double A_van_Driest` (default=`0`):
  van Driest constant for the damping law. (found in file: `nsitetlesfm2.cpp`)
- `double C_smag` (default=`0.18`):
  Smagorinsky constant. (found in file: `nsitetlesfm2.cpp`)
- `double[ndim] G_body` (default= `null vector`): Vector of gravity acceleration (must be constant). (found in file: `nsitetlesfm2.cpp`)
- `int LES` (default=`0`):
  Add LES for this particular elemset. (found in file: `nsitetlesfm2.cpp`)
- `double additional_tau_pspg` (default=`0.`):
  Add to the `tau_pspg` term, so that you can stabilize with a term independently of h. (Mainly for debugging purposes). (found in file: `nsitetlesfm2.cpp`)
- `string axisymmetric` (default=`none`):
  Add axisymmetric version for this particular elemset. (found in file: `nsitetlesfm2.cpp`)
- `int cache_grad_div_u` (default=`0`):
  Cache `grad_div_u` matrix (found in file: `nsitetlesfm2.cpp`)
- `string geometry` (default=`cartesian2d`):
  Type of element geometry to define Gauss Point data (found in file: `nsitetlesfm2.cpp`)
- `int indx_ALE_xold` (default=`1`):
  Pointer to old coordinates in `nodedata` array excluding the first `ndim` values (found in file: `nsitetlesfm2.cpp`)
- `double jacobian_factor` (default=`1.`):
  Scale the jacobian term. (found in file: `nsitetlesfm2.cpp`)
- `int npg` (default=`none`):
  Number of Gauss points. (found in file: `nsitetlesfm2.cpp`)
- `double pressure_control_coef` (default=`0.`):
  Add pressure controlling term. (found in file: `nsitetlesfm2.cpp`)
- `int print_van_Driest` (default=`0`):
  print Van Driest factor (found in file: `nsitetlesfm2.cpp`)
- `double residual_factor` (default=`1.`):
  Scale the residual term. (found in file: `nsitetlesfm2.cpp`)
- `double rho` (default=`1.`):
  Density (found in file: `nsitetlesfm2.cpp`)
- `double shock_capturing_factor` (default=`0`):
  Add shock-capturing term. (found in file: `nsitetlesfm2.cpp`)
- `double tau_fac` (default=`1.`):
  Scale the SUPG and PSPG stabilization term. (found in file: `nsitetlesfm2.cpp`)

- `double tau_pspg_fac` (default=`1.`):
  Scales the PSPG stabilization term. (found in file: `nsitetlesfm2.cpp`)
- `double temporal_stability_factor` (default=`0.`):
  Adjust the stability parameters, taking into account the time step. If the `steady` option is in effect, (which is equivalent to $\Delta t = \infty$) then `temporal_stability_factor` is set to 0. (found in file: `nsitetlesfm2.cpp`)
- `int weak_form` (default=`1`):
  Use a weak form for the gradient of pressure term. (found in file: `nsitetlesfm2.cpp`)

Elemset "`bcconv_ns_fm2`":

- `string geometry` (default=`cartesian2d`):
  Type of element geometry to define Gauss Point data (found in file: `bccnsfm2.cpp`)
- `int weak_form` (default=`1`):
  Use a weak form for the gradient of pressure term. (found in file: `bccnsfm2.cpp`)

Elemset "`wall`":

- `string geometry` (default=`cartesian2d`):
  Type of element geometry to define Gauss Point data (found in file: `wall.cpp`)
- `double rho` (default=`1.`):
  Density (found in file: `wall.cpp`)
- `double y_wall_plus` (default=`25.`):
  The y^+ coordinate of the computational boundary (found in file: `wall.cpp`)

Elemset "`ns_sup_g`":

This element imposes a linearized free surface boundary condition.

- `int LES` (default=`0`):
  Add LES for this particular elemset. (found in file: `nssupg.cpp`)
- `double free_surface_damp` (default=`0.`):
  C_{lf} =`free_surface_set_level_factor` tries to keep the free surface level constant by adding a term $\propto \bar{\eta}$ to the free surface level. (see doc for `free_surface_damp`) The equation of the free surface is

$$(\mathrm{d}\eta/\mathrm{d}t) = w \tag{92}$$

  where η elevation, and w is the velocity component normal to the free surface. We modify this as follows

$$C_{\mathrm{eq}} \frac{\mathrm{d}\eta}{\mathrm{d}t} + C_{\mathrm{lf}}\bar{\eta} - C_{\mathrm{damp}}\,\Delta\eta = w \tag{93}$$

Where $\bar{\eta}$ is the average value of eta on the free surface, and:
C_{damp} = `free_surface_damp` smoothes the free surface adding a Laplacian filter. Note that if only the temporal derivative and the Laplace term are present in (93) then the equation is a heat equation. A null value (which is the default) means no filtering. A high value means high filtering. (Warning: A high value may result in unstability). C_{damp} has dimensions of L^2/T (like a diffusivity). One possibility is to scale with mesh parameters like $h^2/\Delta t$, other is to scale with $h^{1.5}\, g^{0.5}$. Currently, we are using $C_{\text{damp}} = C'_{\text{damp}} h^{1.5} g^{0.5}$ with $C'_{\text{damp}} \approx 2$. (found in file: `nssupg.cpp`)

- `double free_surface_set_level_factor` (default=`0.`):
  This adds a $C_{\text{lf}}\bar{\eta}$ term in the free surface equation in order to have the total meniscus volume constant. (found in file: `nssupg.cpp`)
- `double fs_eq_factor` (default=`1.`):
  C_{eq}=`fs_eq_factor` (see doc for `free_surface_damp` option) is a factor that scales the free surface "rigidity". $C_{\text{eq}} = 1$ (which is the default) means no scaling, a zero value means infinitely rigid (as for an infinite gravity). (found in file: `nssupg.cpp`)
- `string geometry` (default=`cartesian2d`):
  Type of element geometry to define Gauss Point data (found in file: `nssupg.cpp`)
- `int npg` (default=`none`):
  Number of Gauss points. (found in file: `nssupg.cpp`)

7.3 Mesh movement

Sometimes one has a mesh (connectivity and node coordinates) and wants a mesh for a slightly modified domain. If $\mathbf{u}$ is the displacement of the boundaries, then we can pose the problem of finding a mesh topologically equal to the original one, but with a slight displacement of the nodes so that the boundaries are in the new position. This is termed *"mesh relocation"*. One way to do this is to solve an elasticity problem where the displacements at the boundaries are the prescribed displacement of the boundary. This problem has been included by convenience in the Navier-Stokes module, even if at this stage it has little to do with the Navier-Stokes eqs.

Once the displacement is computed by a standard finite element computation, we can compute the new mesh node coordinates by adding simply the computed displacement to the original node coordinate. The elastic constants can be chosen arbitrarily. If an isotropic material is considered, then the unique relevant non-dimensional parameter is the Poisson ratio, controlling the incompressibility of the fictituos material. However, if the distortion is too large, this linear simple strategy can break down, when some element collapses and has a negative Jacobian. A simple idea to fix this is to somewhat rigidize the elastic constants of the fictituos material in order to minimize the distortion of the elements. Designing this nonlinear material behaviour should guarantee a unique solution and a relatively easy way to compute the Jacobian. A possibility is to have an hyperelastic material, i.e. to define an *"energy functional"* $F(\epsilon_{ij})$ as function of the strain tensor. One

should guarantee that this functional is convex, and one should have an easy way to compute its derivatives and second derivatives.

A related approach, implemented in PETSc-FEM, is to compute (for simplices) the transformation from a regular master element to the actual element, and to define the energy functional to be minimized as a function of the associated matrix tensor. If $J_{ij} = (\partial x_i/\partial \xi_j)$ where x_i are the spatial coordinates and ξ_j the master coordinates, then the metric tensor is

$$G_{ij} = \frac{\partial x_k}{\partial \xi_i} \frac{\partial x_k}{\partial \xi_j}. \tag{94}$$

The Jacobian of the transformation may be computed by differentiating the interpolated displacements

$$x_i(\xi_j) = \sum_\mu x_{\mu i} N_\mu(\xi_j), \tag{95}$$

with respect to the master coordinates, so that

$$J_{ij} = \frac{\partial x_i}{\partial \xi_j} = \sum_\mu x_{\mu i} \frac{\partial N_\mu}{\partial \xi_j}. \tag{96}$$

We want to compute the distortion function to be minimized as a function of the eigenvalues of the metric tensor G_{ij}. The eigenvalues $\mathbf{v}_q$ of $\mathbf{G}$ are such that

$$G_{ij} v_{qj} = \lambda_q v_{qj} \tag{97}$$

so that,

$$\lambda_q = v_{qi}\, G_{ij}\, v_{qj} \tag{98}$$

(no sum over q) since the $\mathbf{v}_q$ are orthogonal. Now if the functional F is a function of the element node coordinates through the eigenvalues λ_q then we can compute the new coordinates x' as

$$\frac{\partial F}{\partial x_{\mu j}} = 0. \tag{99}$$

A possible distortion functional is

$$F(\{\lambda_q\}) = (\Pi_q \lambda_q)^{-2/n_{\mathrm{dim}}} \sum_{qr} (\lambda_q - \lambda_r)^2 \tag{100}$$

This functional has several nice features. Is minimal whenever all the eigenvalues are equal (the distortion is minimal). It is non-dimensional, so that an isotropic dilatation or contraction doesn't produce a change in the functional. The non-linear problem (99) can be solved by a Newton strategy, by computing the first and second derivatives of F with respect to the node displacements (the residual and Jacobian of the system of equations) by finite difference approximations. However, this turns to be too costly in CPU time for the second derivatives, since we should compute for each second derivative three evaluations of the functional, and there are $n_{\mathrm{en}}(n_{\mathrm{en}}+1)/2$ (where $n_{\mathrm{en}} = n_{\mathrm{dof}} n_{\mathrm{el}}$ is the number of unknowns per element, n_{el} is the number of nodes per element, and n_{dof} the number of unknowns per node, here $n_{\mathrm{dof}} = n_{\mathrm{dim}}$) second derivatives to compute. Each evaluation of the functional amounts to the computation of the tensor metrics G and to solve the associated eigenvaue problem, so that an analytical expression to, at least, the

first derivatives, is desired. The derivatives of the distortion functional can be computed as

$$\frac{\partial F}{\partial x_j} = \frac{\partial F}{\partial \lambda_q} \frac{\partial \lambda_q}{\partial x_{\mu j}} \tag{101}$$

and then the derivatives with respect to the eigenvalues can be computed still numerically, while we will show that the derivatives of the eigenvalues with respect to the node coordinates (which are the most expensive part) can be computed analytically. In this way, we can compute the derivatives of the functional with the solution of only one eigenvalue problem. The second derivatives can be computed similarly as

$$\frac{\partial^2 F}{\partial x_{\mu i} x_{\nu j}} = \frac{\partial^2 F}{\partial \lambda_q \lambda_r} \frac{\partial \lambda_q}{\partial x_{\mu i}} \frac{\partial \lambda_r}{\partial x_{\mu j}} + \frac{\partial F}{\partial \lambda_q} \frac{\partial^2 \lambda_q}{\partial x_{\mu i} x_{\nu j}}. \tag{102}$$

The first and second derivatives of F with respect to the eigenvalues are still computed numerically, whilst the second derivatives of the eigenvalues can be computed by differentiating numerically the first derivatives. This amounts to $O(n_{\text{en}})$ eigenvalue problems for computing the first and second derivatives of the distortion functional (the residual and Jacobian). This cost may be further reduced by noting that the eigenvalues are invariant under rotations and translations, and simply scaled by a dilatation or contraction, so that, from the $n_{\text{en}} = n_{\text{dim}}(n_{\text{dim}} + 1)$ displacements, only $n_{\text{dim}}(n_{\text{dim}} + 1)/2 - 1$ should be really computed, but this is not implemented yet. For tetras in 3D this implies a reduction from 12 distortion functional computation to only 5.

We will show now how the derivatives of the eigenvalues are computed analytically. It can be shown that

$$\frac{\partial \lambda_q}{\partial x_{\mu k}} = v_{qi} \frac{\partial G_{ij}}{\partial x_{\mu k}} v_{qj}. \tag{103}$$

Note that this is as differentiating (98) but only keeping in the right hand side the change in the matrix G and discarding the rate of change of the eigenvectors. It can be shown that the conttribution of the other two terms is an antisymmetric matrix, so that the contibution to the rate of change of the eigenvalue is null. Now, from (94)

$$\frac{\partial G_{ij}}{\partial x_{\mu l}} = J_{li} \frac{\partial N_\mu}{\partial \xi_j} + J_{lj} \frac{\partial N_\mu}{\partial \xi_i} \tag{104}$$

so that

$$\frac{\partial \lambda_q}{\partial x_{\mu l}} = (v_{ql}\, J_{li}) \left(\frac{\partial N_\mu}{\partial \xi_j} v_{qj} \right), \tag{105}$$

which is the expression used.

8 Tests and examples

In this section we describe some examples that come with PETSc-FEM. They are intentionally small and sometimes almost trivial, but they are very light-weight and can be run in a short time (at most some minutes). They are put in the `test` directory and can be run with `make tests` command. The purpose of this tests is to check the corect behaviour of PETSc-FEM, but also they can serve people to understand the program.

8.1 Flow in the anular region between to cylinders

File: `sector.dat`. This example tests periodic boundary conditions. The mesh is a sector of circular strip ($2.72 < r < 4.48$, $0 < \theta < \pi/4$) and we impose. The governing equation is $\Delta \mathbf{u} = 0$, where $\mathbf{u}$ is a velocity vector of two components. On the internal radius we impose $\mathbf{u} = 0$ and $\mathbf{u} = \mathbf{t}$ where $\mathbf{t}$ is the tangent vector. On the radii $\theta = 0$ and $\theta = \pi/4$ we impose periodic boundary conditions so that it is close to flow in the region between two cylinders, with the internal cylinder fixed and the external one rotating with velocity 1. But the operator *is not* the Stokes operator. However in this case the flow for this operator is divergence free and the gradient of pressure has no component in the angular direction, so that the solution *do* coincide with the Stokes solution.

In the output of the test (sector.sal), we check that the solution is aligned with the angular direction $u_x = -u_y$ at the outlet section ($\theta = \pi/4$).

8.2 Flow in a square with periodic boundary conditions

File: `lap_per.dat` This is similar to example "sector.dat" but in a region that is the square $-1 < x, y < +1$. $\mathbf{u}$ is imposed on all sides, $|u| = 1$ and in the tangential direction in the counter-clockwise sense, i.e. $\mathbf{u} = (0, \pm 1)$ in $x = \pm 1$, $\mathbf{u} = (\mp 1, 0)$ in $y = \pm 1$. By symmetry we can solve it in $1/4$-th of the domain, i.e. in the square $x, y > 0$ (We could solve it also in $1/8$-th of the region, the triangle $y > 0$, $y < x$, $x < 1$).

8.3 The oscilating plate problem

File: `oscplate.dat` This is the flow produced between two infinite plates when one is at rest and the other oscillating with amplitude A and frequency ω (see figure 7). This serves to test a problem with temporal dependent boundary conditions. The problem is one-dimensional and the resulting field is $u = 0$, $p = \text{cnst}$ and $v = v(x)$. We set parameters lenght between plates $L = 1$, viscosity $\nu = 1$ and we model the problem with a strip $0 \le x \le 1$, $0 \le y \le 0.1$ and set periodic boundary conditions between $y = 0.1$ and $y = 0$. At the plate at rest ($x = L$) we set $u, p = 0$ and at $x = 0$ and at the moving plate ($x = 0$) we set $u = \omega A \sin(\omega t)$. The analytic solution can be found easily. The equation for v is

$$\frac{\partial v}{\partial t} = \nu \frac{\partial^2 v}{\partial x^2} \tag{106}$$

with boundary conditions $v(0) = A\omega \sin(\omega t)$ and $v(L) = 0$. The solution can be found by searching for solutions of the form

$$v = \mathrm{e}^{\mathrm{i}\omega t + \lambda x} \tag{107}$$

Replacing in (106) we arrive at the characteristic equation

$$\mathrm{i}\omega = \nu \lambda^2 \tag{108}$$

whose solutions are

$$\lambda = \pm \frac{1 + \mathrm{i}}{\sqrt{2}} \sqrt{\omega/\nu} = \pm \lambda_+ \tag{109}$$

Setting the boundary conditions, the solution is

$$v(x) = \mathrm{Im}\left\{ \mathrm{e}^{\mathrm{i}\omega t} \, \frac{\mathrm{e}^{\lambda_+(x-L)} - \mathrm{e}^{-\lambda_+(x-L)}}{\mathrm{e}^{-\lambda_+ L} - \mathrm{e}^{\lambda_+ L}} \right\} \tag{110}$$

In the example we set $\nu = 100$, $\omega = 2000\pi \approx 6283$. We choose to have 16 time steps in one period so that $Dt = 6.2 \times 10^{-5}$. The resulting profile velocity is compared with the analytical one in figure 8.4s

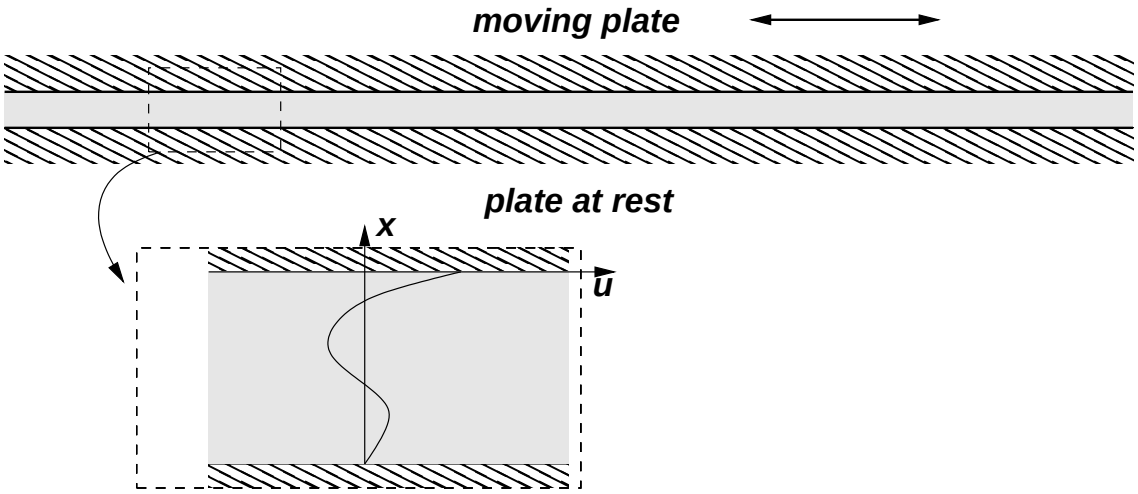


Figure 7: Oscillating plates.

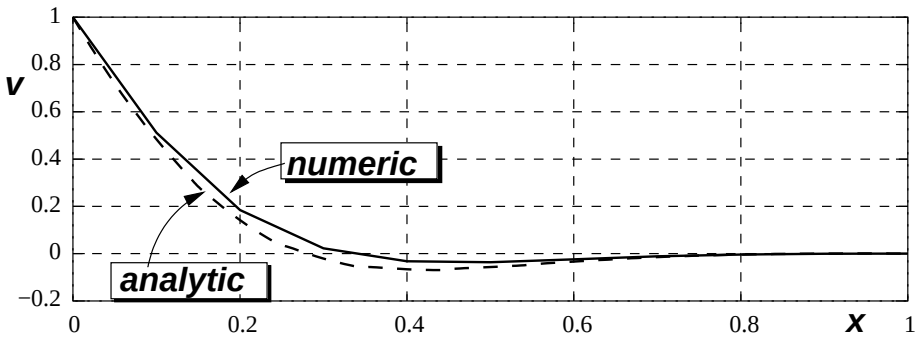


Figure 8: Velocity profile for the oscilating plate problem.

8.4 Linear advection-diffusion in a rectangle

File: `sine.epl` This is an example for testing the `advdif` module. The governing equations are

$$\begin{aligned}
&\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} = 0, \text{ in } 0 \le x \le L_x, |y| < \infty, t > 0 \\
&\phi = A\,\cos(\omega t)\,\cos(ky), \text{ at } x = 0, t > 0 \\
&\frac{\partial \phi}{\partial n} = 0, \text{ at } x = L_x, t > 0 \\
&\phi = 0, \text{ at } 0 \le x \le L_x, |y| < \infty, t = 0
\end{aligned} \tag{111}$$

As the problem is perdiodic in the y direction we can restrict the analysis to one quart wave-length, i.e. if $\lambda_y = 2\pi/k$, $L_y = \lambda/4$ then the above problem is equivalent to

$$\begin{aligned}
\frac{\partial \phi}{\partial t} + u\frac{\partial \phi}{\partial x} &= D\Delta\phi, \text{ in } 0 \le x \le L_x,\ 0 < y < L_y, t > 0 \\
\phi = A\,\cos(\omega t)\,\cos(ky), &\text{ at } x = 0, t > 0 \\
\frac{\partial \phi}{\partial n} &= 0, \text{ at } x = L_x, t > 0 \\
\phi &= 0, \text{ in } 0 \le x \le L_x,\ 0 < y < L_y, t > 0 \\
\phi &= 0, \text{ at } y = L_y \\
\frac{\partial \phi}{\partial n} &= 0, \text{ at } y = 0
\end{aligned} \tag{112}$$

We can find the solution proposing a solution of the form

$$\phi \propto \mathrm{e}^{\beta x + \mathrm{i}\omega t)}$$

Replacing in (112) we arrive to a characteristic equation in k_x of the form

$$\mathrm{i}\,\omega + \beta u = -k_y^2 + \beta^2 \tag{113}$$

This is a quadratic equation in β and has two roots, say $\beta - 12$. The general solution is

$$\phi(x, y, t) = \mathrm{Re}\left\{\left(c_1 \mathrm{e}^{\beta_1 x} + c_2 \mathrm{e}^{\beta_2 x}\right)\mathrm{e}^{\mathrm{i}\omega t}\right\}$$

9 The FastMat2 matrix class

9.1 Introduction

Finite element codes usually have two levels of programming. In the outer level a large vector describes the *"state"* of the physical system. Usually this vector has as many entries as the number of nodes times the number of fields minus the number of fixations (i.e. Dirichlet boundary conditions). This vector can be computed at once by assembling the right hand side and the stiffness matrix in a linear problem, iterated in a non-linear problem or updated at each time step through solution of a linear or non-linear system. The point is that, at this outer level, you perform global assemble operations that build this vector and matrices. At the inner level, you perform a loop over all the elements

in the mesh, compute the vector and matrix contributions of each element and assemble them in the global vector/matrix. From one application to another, the strategy at the outer level (linear/non-linear, steady/temporal dependent, etc...) and the physics of the problem that defines the FEM matrices and vectors may vary.

The FastMat2 matrix class has been designed in order to perform matrix computations at the element level. It is assumed that we have an outer loop (usually the loop over elements) that is executed many times, and at each execution of the loop a series of operations are performed with a rather reduced set of local vectors and matrices. There are many matrix libraries but often the performance is degraded for small dimensions. Sometimes performance is good for operations on the whole matrices, but it is degraded for operations on a subset of the elment of the matrices, like columns, rows, or individual elements. This is due to the fact that acessing a given element in the matrix implies a certain number of arithmetic operations. Otherwise, we can copy the row or column in an intermediate object, but then there is an overhead due to the copy operations.

The particularity of FastMat2 is that at the first execution of the loop the address of the elements used in the operation are cached in an internal object, so that in the second ans subsequent executions of the loop the addresses are retrieved from the cache.

9.2 Example

Consider the following simple example. We are given a 2D finite element composed of triangles, i.e. an array `xnod` of $2 \times N_{\text{nod}}$ doubles with the node coordinates and an array `icone` with $3 \times n_{\text{elem}}$ elements with node connectivities. For each element $0 < j < n_{\text{elem}}$ its nodes are stored at `icone[3*j+k]` for $0 \leq k \leq 2$. We are required to compute the maximum and minimum value of the area of the triangles. This is a computation which is similar to those found in FEM analysis. For each element we have to load the node coordinates in local vectors $\mathbf{x}_1$, $\mathbf{x}_2$ and x_3, compute the vectors along the sides of the elements $\mathbf{a} = \mathbf{x}_2 - \mathbf{x}_1$ and $\mathbf{b} = \mathbf{x}_3 - \mathbf{x}_1$. The area of the element is, then, the determinant of the 2×2 matrix $\mathbf{J}$ formed by putting $\mathbf{a}$ and $\mathbf{b}$ as rows.

The FastMat2 code for the computations is like this

```
Chrono chrono;
FastMat2 x(2,3,2),a(1,2),b(1,2),J(2,2,2);
FastMatCacheList cache_list;
FastMat2::activate_cache(&cache_list);
// Compute area of elements
chrono.start();
for (int ie=0; ie<nelem; ie++) {
  FastMat2::reset_cache();
  for (int k=1; k<=3; k++) {
    int node = ICONE(ie,k-1);
    x.ir(1,k).set(&XNOD(node-1,0)).rs();
  }
  x.rs();
  a.set(x.ir(1,2));
  a.rest(x.ir(1,1));
```

```
    b.set(x.ir(1,3));
    b.rest(x.ir(1,1));

    J.ir(1,1).set(a);
    J.ir(1,2).set(b);

    double area = J.rs().det()/2.;
    total_area += area;
    if (ie==0) {
      minarea = area;
      maxarea = area;
    }

    if (area>maxarea) maxarea=area;
    if (area<minarea) minarea=area;
  }
  printf("total_area %g, min area %g,max area %g, ratio: %g\n",
 total_area,minarea,maxarea,maxarea/minarea);
  printf("Total area OK? : %s\n",
 (fabs(total_area-1)<1e-8 ? "YES" : "NOT"));
  double cpu = chrono.elapsed();
  FastMat2::print_count_statistics();
  printf("CPU: %g, number of elements: %d\n"
 "rate: %g [sec/Me], %g Mflops\n",
 cpu,nelem,cpu*1e6/nelem,
 nelem*FastMat2::operation_count()/cpu/1e6);
  FastMat2::void_cache();
  FastMat2::deactivate_cache();
```

Calls to the static members (those starting with `FastMat2::`) are related with the caching manipulation and will be discussed later. Matrix are dimensioned in line 2, the first argument is the number of dimensions, and then follow the dimensions, for instance `FastMat2 x(2,3,2)` defines a matrix which 2 indices ranging from 1 to 3, and 1 to 2 respectively. The rows of this matrix will store the coordinates of the local nodes to the element. FastMat2 matrices may have any number of indices. Actually the library is compiled for a maximum number of indices (10 by default. This limit may be modified by redefining variable `$max_arg` in script `readlist.eperl` and recompile.) Also they can have zero dimensions, which stands for scalars.

9.2.1 Current Matrix view

In lines 9 to 12 the coordinates of the nodes are loaded in matrix `x`. The underlying philosophy in FastMat2 is that you can set *"views"* of the matrix without actually made any copies of the underlying values. For instance the operation `x.ir(1,k)` (for *"index restriction"*) sets a view of `x` so that index 1 is restricted to take the value k reducing

in one the number of dimensions of the matrix. As `x` has two indices, the operation `x.ir(1,k)` gives a matrix of dimension one consisting in the k-th row of x. A call without arguments like in `x.ir()` cancels the restriction. Also, the function `rs()` (for *"reset"*) cancels the actual view.

9.2.2 Set operations

The operation `a.set(x.ir(1,2))` copies the contents of the argument `x.ir(1,2)` in `a`. Also we can use `x.set(y)` with `y` a Newmat matrix (`Matrix y`) or an array of doubles (`double *y`).

9.2.3 Dimension matching

The `x.set(y)` operation requires that `x` and `y` have the same "viewed" dimensions. As the `.ir(1,2)` operation restricts index to the value of 2, `x.ir(1,2)` is seen as a row vector of size 2 and then can be copied to a. If the "viewed" dimensions don't fit then an error is issued.

9.2.4 Automatic dimensioning

In the example, `a` has been dimensioned at line 2, but most operations perform the dimensioning if the matrix has not been already dimensioned. For instance, if at line 2 we would declared `FastMat2 a` only, without specifying dimensions, then at line 14, the matrix is created and dimensioned taking the dimensions from the argument. The same applies to `set(Matrix &)` but not to `set(double *)` since in this last case the argument (`double *`) don't posses information about his dimensions. Other operations that define dimensions are products and contraction operations.

9.2.5 Concatenation of operations

Many operations return a reference to the matrix (return value `FastMat2 &`) so that operations may be concatenated as in `A.ir(1,k).ir(2,j)`.

9.3 Caching the adresses used in the operations

If caching is not used the performance of the library is poor, typically one to two orders of magnitude slower than the cached version, and the cached version is very fast, in the sense that almost all the CPU time us spent in performing multiplications and additions, and negligible CPU time is spent in auxiliary operations.

The idea with caches is that they are objects (`class FastMatCache`) that store the adresses needed for the current operation. In the first pass through the body of the loop a cache object is created for each of the operations, and stored in a list of class `FastMatCacheList`. This list is basically an STL vector of pointers to cache objects. When the body of the loop is executed the second time and the following, the adresses are not needed to be recomputed but they are read from the cache instead. The use of the cache is rather automatic and requires little intervention by the user but in some cases the position in the cache-list can get out of sync with respect to the execution of the operations and severe errors may occur.

The typical use can be seen in the example shown in section §9.2. First we have to create a `FastMatCacheList` object as shown in line 3 and activate it with as in line 4. The outer loop here is the loop over elements. For the second and following executions of the body of the loop, you must "rewind" the cache-list, this is done in line 8 with the `FastMat2::reset_cache()` operation (see figure 9).

The `deactivate_cache()` call at line 43 causes that subsequent operations after this line will not be cached. This call is required if subsequent calls to FastMat2 cached operations will be executed. Otherwise there will be an error since those posterior calls will not have a corresponding cache in the cache-list.

The `void_cache()` call deletes the cache claiming the space used by it. It's not required but it is a good idea in order to save memory space. It may be required if the loop will be called with other dimensions. For instance a FEM loop may be called first for triangles and then for quadrangles, and in this two calls the dimensions of the involved matrices are different.

The general layout of a code section which uses caching is like this

```
...           // Initialization. Not cached operations.

FastMatCacheList cache_list;           // Cache-list object
FastMat2::activate_cache(&cache_list); // activates the cache

for (j=0; j<N; j++) {    // Outer loop. N very large number

  FastMat2::reset_cache();            // Rewinds the cache list

  op_1;
  op_2;
  op_3;                               // Cached operations
  ...
  op_N;


}                        // End of outer loop

FastMat2::void_cache();               // free memory
FastMat2::deactivate_cache();         // deactivates cache
```

9.3.1 Branching

Caching { is straightforward if the sequence of operations are executed linearly, with the same local variables, without branching or looping. Special operations have to be executed if there are branching conditions (`if` sentences) that alter the order of execution of the operations in the loop. For instance an "if" sequence like this

```
%
....
op_prev_1;               // operations previous to the branch
```

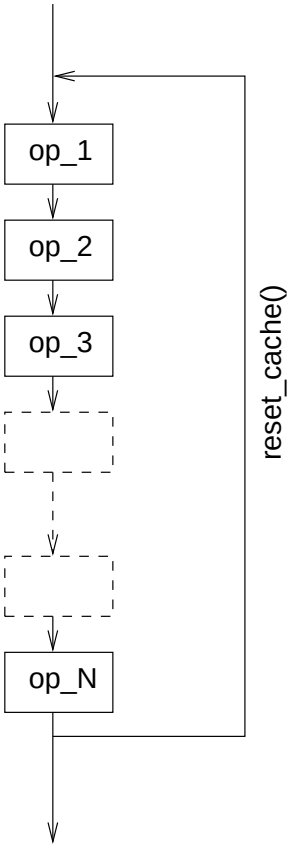


Figure 9: Cache of operations for linear segment of code

```
op_prev_2;
...
op_prev_k;
if (<condition 0>) {
  op_0_1;
  op_0_2;                // conditional block for branch 0
  ...
  op_0_N;
} else if (<condition 1>) {
  op_1_1;
  op_1_2;                // conditional block for branch 1
  ...
  op_1_N;
} else {
  op_1_1;
  op_1_2;               // conditional block for the ‘else’ branch
  ...
  op_1_N;
}
```

```
....
op_pos_1;                // operations posterior to the branch
op_pos_2;
...
op_pos_k;
```

If branch '0' is followed in the first execution of the block, then the cache will look like that shown in figure 11. When in a subsequent execution of the loops another branch is

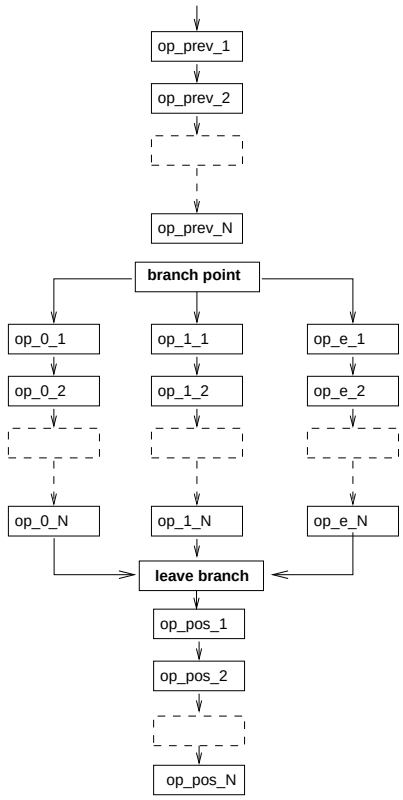


Figure 10: Cache list produced when branch 0 is chosen.

chosen, say branch 1, then when reading trying to execute operation `op_1_1` the library will find in the cache list a cache corresponding to operation `op_0_1`.

This is solved by creating *"branch points"* in the cache list and choosing the appropriate branch as shown in the following code (see figure 10):

```
%
....
op_prev_1;               // operations previous to the branch
op_prev_2;
...
op_prev_k;
FastMat2::branch();
if (<condition 0>) {
```

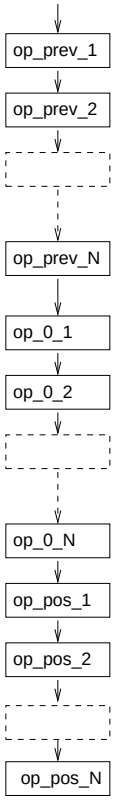


Figure 11: Cache list produced when branch 0 is chosen.

```
  FastMat2::choose(0);
  op_0_1;
  op_0_2;                // conditional block for branch 0
  ...
  op_0_N;
} else if (<condition 1>) {
  FastMat2::choose(1);
  op_1_1;
  op_1_2;                // conditional block for branch 1
  ...
  op_1_N;
} else {
  FastMat2::choose(<N>);
  op_e_1;
  op_e_2;               // conditional block for the 'else' branch
  ...
  op_e_N;
}
FastMat2::leave();
....
```

```
op_prev_1;                // operations posterior to the branch
op_prev_2;
...
op_prev_k;
```

The `branch()` call tells the library that several branchs will start from there and a branch point is created. Then each conditional block code must start with

`choose(<j>)`

where <j> is a number that must be unique among all other branches. Finally when leaving all the branches we must call `leave()` in order to tell the library that the mainstream of the cache list must be retaken. Branches can be nested at any level.

The call to branching is not needed if the "execution path" is the same for all executions of the loop. This usually happens when the condition refer to some global option that is uniform over all elements. For instance if branch '0' corresponds to "include turbulence model A" and branch '0' to model B, then the same branch is executed for all the elements and there is no need to call the static functions.

9.3.2 Loops executed a non constant number of times

Another special case is when there are loops inside the body of the outer loop. Note that no special branching is needed in general if the loop is executed a fixed number of times, since the sequence of operations is not altered from one execution to another. For instance consider the following piece of code

```
//   Case A. Inner code executed a fixed number of times
...
for (int k=0; k<N;k++) {              // N very large - Outer loop
  block_before;
  for (int ll=0; ll<3; ll++) {
    inner_block;                      // Operations that act on the
                                      // same matrices.
  }
  block_after;
}
```

Then the cache list generated in the first execution of the loop will be

```
block_before
inner_block
inner_block
inner_block
block_after
```

and this is OK since the number of times `inner_block` is executed is always the same. If the operations that are performed inside the loop are the the same for all executions of the loop but are executed an irregular number of times, then we can use a sequence as follows

```
//   Case B. Inner code executed a variable number of times
FastMatCachePosition cp1;
...
for (int k=0; k<N;k++) {              // N very large - Outer loop
  block_before
  FastMat2::get_cache_position(cp1);
  int n=irand(1,5);
  for (int ll=0; ll<n; ll++) {
    FastMat2::jump_to(cp1);
    inner_block;                      // Operations that act on the
                                      // same matrices.
  }
  FastMat2::resync_was_cached();
  block_after;
```

Here the number of times the inner block is executed may vary randomly from 1 to 5. (`irand(m,n)` returns an integer number randomly distributed between `m` and `n`.) The `FastMatCachePosition` class objects store the position of the actual computation in the cache list. So that the call to `jump_to()` at the start of the loop restarts the position in the cache to the desired one. After leaving the loop we call to `resync_was_cached()` in order to resync the cache list.

This is OK if the inner loop is executed at least once the in the first execution of the outer loop. If it happens that in the first execution of the loop the inner loop is not entered, then the cache list will contain `block_before,block_after` and when the inner block will be entered in subsequent executions of the loop and error will arise since there will be missimng caches.

To fix this we have to combine this with branching as here

```
FastMatCachePosition cp1;
for (int k=0; k<N;k++) {    // N very large - Outer loop
  ....                      // Previous block
  FastMat2::branch();       // Allows conditional execution
  FastMat2::choose(0);

  FastMat2::get_cache_position(cp1);
  n=irand(0,5);
  if (k==0) n=0; // This is the critical case.
                 // n=0 the first execution
              // of the loop.
  for (int ll=0; ll<n; ll++) {
    FastMat2::jump_to(cp2);
    ...                     // inner_block
  }
  FastMat2::resync_was_cached();
  FastMat2::leave();
  ...                       // posterior block
}
```

Off course, if the number of times the inner loop is executed is very large, and the most time consuming part is the execution of this loop, then it may be convenient to choose this loop as the "outer" one.

9.3.3 Masks can't traverse branches

Another restriction is that if branching is used, the mask that is active at a certain `FastMat2` cached operation must be the same independently of the path that the code have followed, for instance consider the following code

```
FastMat2 a,b,c;
// resize and set 'a,b,c'
for (int j=0; j<N; j++) {
  FastMat2::branch();
  if (condition) {
    FastMat2::choose(0);
    a.is(...).ir(...);      // (B)
    // operate on masked 'a'
  }
  FastMat2::leave();
  c.prod(a,b);              // (A) Wrong! 'a' may have or
                            // not the mask set
}
```

When the code reaches the `prod()` method at line `(A)`, it can have executed or not the block inside the `if`, so that the mask set in line `(B)` may or may not be active at line `(A)`. This is clearly an error, and to avoid it the safest way is to always reset the masks at the outlet of a branched block like in line `(C)` as follows.

```
FastMat2 a,b,c;
// resize and set 'a,b,c'
for (int j=0; j<N; j++) {
  FastMat2::branch();
  if (condition) {
    FastMat2::choose(0);
    a.is(...).ir(...);      // (B)
    // operate on masked 'a'
    a.rs();                 // (C)
  }
  FastMat2::leave();
  c.prod(a,b);              // (A) OK! 'a' has not mask.
}
```

9.3.4 Efficiency

As we mentioned before, When caching is enabled there is a gain in speed of ten to one hundreth, and the library is very performant. Of course, the first execution of loop is not cached and represents and overhead that has to be amortized by executing the loop in

cached mode many times. The average speed increases when the number of executions of the loop is increased. The cut point, i.e. the number of executions of the loop for which the excution speed falls to one half the speed obtained for very large number o execution is currently between 10 and 30, so that for loops larger than 200 the overhead time spent in building the caches is negligible.

Another issue is the memory required by the caches. First there is some space required by the caches themselves and then, there is a copy of the addresses of the elements involved. For instance in a `a.set(b)` operation with `a` and `b` of size $n \times m$, say, we have to store $2mn$ addresses. Usually this overhead in memory requirement is negligible, since the amount of variables and operations needed in the element routines are very small as compared with the size of the problem itself. However, some care must be taken when caching large inner loops. For instance in code A, section §9.3.2, if the inner loop is executed a constant, but very large, number M of times, then the amount of the cache required is proportional to M. Then, even if, as discussed before, no operations like those used in code B are required, it may be adviceable to spend some time in insert these calls in order to reduce memory cache and overhead time. Again, in the limit of M very large, it will be more convenient to choose this loop as the "outer" one.

9.4 Synopsis of operations

9.4.1 One-to-one operations

These are operations that take one FastMat2 argument as in
`FastMat2& add(const FastMat2 & A)`. The operations are from one element of `A` to the corresponding element in `*this`.
The one-to-one operations implemented so far are

- `FastMat2& set(const FastMat2 & A)` Copy matrix
- `FastMat2& add(const FastMat2 & A)` Add matrix
- `FastMat2& rest(const FastMat2 & A)` Substract a matrix
- `FastMat2& mult(const FastMat2 & A)` Multiply (element by element) (like Matlab `.*`).
- `FastMat2& div(const FastMat2 & A)` Divide matrix (element by element, like Matlab `./`).
- `FastMat2& axpy(const FastMat2 & A, const double alpha)` Axpy operation (element by element): `(*this)` = alpha * A+

9.4.2 In-place operations

These operations perform an action on all the elements of a matrix.

- `FastMat2& set(const double val=0.)` Sets all the element of a matrix to a constant value
- `FastMat2& scale(const double val)` Scale by a constant value

- `FastMat2& add(const double val)` Adds constant val
- `FastMat2& fun(scalar_fun_t *function)` Apply a function to all elements
- `FastMat2& fun(scalar_fun_with_args_t *function, void *user_args)` Apply a function with optional arguments to all elements

9.4.3 Generic "sum" operations (sum over indices)

These operations perform some operation an all the indices of a given dimension resulting in a matrix which has less number of indices. It's a generalization of the `sum/max/min` operations in Matlab that returns the specified operation per columns, resulting in a row vector result (one element per column). Here you specify a number of integer arguments, in such a way that

- if the j-th integer argument is positive it represents the position of the index in the resulting matrix, otherwise
- if the j-th argument is -1 then we perform the specified operation (sum/max/min etc...) over all this index.

For instance if we declare `FastMat2 A(4,2,2,3,3)` then `B.sum(A,-1,2,1,-1)` means

$$B_{ij} = \sum_{k=1..2,l=1..3} A_{kjil}, \quad \text{for} \quad i = 1..3,\ j = 1..2 \tag{114}$$

These operation can be extended to any binary associative operation. So far we have implemented the following

- `FastMat2& sum(const FastMat2 & A, const int m=0, ...)` Sum over all selected indices
- `FastMat2& sum_square(const FastMat2 & A, const int m=0, ...)` Sum of squares over all selected indices
- `FastMat2& sum_abs(const FastMat2 & A, const int m=0, ...)` Sum of absolute values all selected indices
- `FastMat2& min(const FastMat2 & A, const int m=0, ...)` Minimum over all selected indices
- `FastMat2& max(const FastMat2 & A, const int m=0, ...)` Maximum over all selected indices
- `FastMat2& min_abs(const FastMat2 & A, const int m=0, ...)` Min of absolute value over all selected indices
- `FastMat2& max_abs(const FastMat2 & A, const int m=0, ...)` Max of absolute value over all selected indices

9.4.4 Sum operations over all indices

When the sum is over all indices the resulting matrix has zero dimensions, so that it is a scalar. You can get this scalar by creating an auxiliar matrix (with zero dimensions) casting with operator `double()` as in

```
FastMat2 A(2,3,3),Z;

...  // assign elements to A

double a = double(Z.sum(A,-1,-1));
```

or using the `get()` function

```
double a = Z.sum(A,-1,-1).get();
```

without arguments, which returns a double. In addition there is for each of the previous mentioned "generic sum" function a companion function that sums over all indices. The name of this function is obtained by appending `_all` to the generic function

```
double a = A.sum_square_all();
```

The list of these functions is

- `double sum_all() const` Sum over all indices
- `double sum_square_all() const` Sum of squares over all indices
- `double sum_abs_all() const` Sum of absolute values over all indices
- `double min_all() const` Minimum over all indices
- `double max_all() const` Maximum over all indices
- `double min_abs_all() const` Minimum absolute value over all indices
- `double max_abs_all() const` Maximum absolute value over all indices

9.4.5 Export/Import operations

These routines allow to convert matrices from or to arrays of doubles and Newmat matrices

- `FastMat2& set(const Matrix & A)` Copies to argument from Newmat matrix
- `FastMat2& set(const double *a)` Copy from array of doubles
- `const FastMat2& export(double *a) const` exports to a double vector
- `FastMat2& export(double *a)` exports to a double vector
- `const FastMat2& export(Matrix & A) const` Exports to a Newmat matrix
- `const FastMat2& export(Matrix & A) const` Exports to a Newmat matrix

9.4.6 Static cache operations

These routines control the use of the cache list.

- `static void activate_cache(FastMatCacheList *cache_list_=NULL)` Activates use of the cache
- `static(void)` Deactivates use of the cache
- `static void reset_cache(void)` Resets the cache
- `static void void_cache(void)` Voids the cache
- `static void branch(void)` Creates a branch point
- `static void choose(const int j)` Follows a branch
- `static void leave(void)` Leaves the current branch
- `static double operation_count(void)` Computes the total number of operations in the cache list
- `static void print_count_statistics()` Print statistics about the number of operations of each type in the current cache list

10 Hooks

Hooks are functions that you pass to the program and, then, are executed at particular points, called *"hook-launching points"*, in the execution of the program. The particular hook-launching points may depend on the application but, in order to fix ideas, for the Navier-Stokes module and the Advective-Diffusive module the standard hooks are:

- `init:` To be executed once, at the start of the program.
- `time_step_pre:` To be executed before the time step calculation.
- `time_step_post:` To be executed after the time step calculation.
- `close:` To be executed once, at the end of the program.

There are *"built-in"* hooks included in the modules, for instance the DX hook that is in charge of communicating with the DX visualizations program or the *"shell-hook"* that allows you to execute shell commands, but you can also define your own hooks that are defined in a C++ piece of code, compiled and dynamically loaded at run-time. You can do almost anything with your hooks, for instance you can compress the result files, perform file manipulation, launch visualization with other software like GMV. Also, hooks are useful for communicating between different instances of PETSc-FEM. For instance, if you want to couple a inviscid external flow with an internal viscous flow, then you can run a PETSc-FEM instance for each region and perform the communication between the different regions with hooks. The DX hook is explained in the DX section (see §13) and we will explain here how to write and use dynamically loaded hooks and the shell hook.

The hook concept has been borrowed from GNU Emacs.

10.1 Launching hooks. The hook list

In order to activate a hook you first have to add a for each hook a pair of strings to the `hook_list`, namely the type pf hook and the name of the hook. This last one is a unique identifier that makes that hook unique.

`hook_list` *<hook-type-1> <hook-name-1> <hook-type-2> <hook-name-2> ...*

For instance:

```
hook_list shell_hook compress     \
          dx_hook    my_dx_hook   \
          dl_hook    coupling_hook
```

Here we added a shell hook that probably will compress some files during execution, the DX hook in order to visualize, and a dynamically linked hook that will couple the run with another program. Each hook will after take their own options from special options in the table.

The `hook_list` entry must be unique, so that you have to group all your hooks in a *single* `hook_list` entry. This is not limiting, because you can add as much hooks as you want, but it is rather syntactically cumbersome, because you end up with a long string. Also it becomes difficult to comment out some hooks while keeping others.

The hooks are executed in the order as you entered them in the hook list, so that in the previous case you will have, at init time, the `init` part of the `compress` hook to be executed *before* the init part of the `my_dx_hook` and finally the init part of the `coupling_hook`.

10.2 Dynamically loaded hooks

The easiest way to code a dynamically loadable hook is with a class. You need to include the corresponding headers `hook.h` and `dlhook.h`, and the class may define the hook functions for all, some or none of the hook-launching points. Consider for instance the following "Hello world!' hook, that prints the message at the corresponding points in the program.

```
#include <src/hook.h>
#include <src/dlhook.h>

class hello_world_hook {
public:
  void init(Mesh &mesh_a,Dofmap &dofmap,
    TextHashTableFilter *options,const char *name) {
    printf("Hello world!I'm in the \"init\" hook\n");
  }
  void time_step_pre(double time,int step) {
    printf("Hello world!I'm in the \"time_step_pre\" hook\n");
  }
  void time_step_post(double time,int step,
      const vector<double> &gather_values) {
```

```
    printf("Hello world!I'm in the \"time_step_post\" hook\n");
  }
  void close() {
    printf("Hello world!I'm in the \"close\" hook\n");
  }
};

DL_GENERIC_HOOK(hello_world_hook);
```

You can use almost any conceivable C/C++ library within your hooks. Take into account that the program may be called in a parallel environment so, for instance, if you will compress a certain file, then you should take care of doing that *only* at the master process by, e.g., enclosing the code with a `if (!MY_RANK) { ... }` construct.

10.3 Shell hook

The *shell-hook* allows the user to execute a certain action at the hook-launching points by simply writing shell commands.

`hook_list shell_hook` <*name*>
<*name*> <*shell-command*>

For instance

```
hook_list shell_hook hello
hello "echo Hello world"
```

In this case, PETSc-FEM will issue the echo command at each of the launching points. If you want to issue more complex commands, then perhaps it's a better idea to bundle them in a script and then execute the script from the hook:

```
hook_list shell_hook hello
hello "my_script_hook"
```

where you have previously written a `my_script_hook` script with something like

```
#!bin/bash
## This is 'my_script_hook' file

echo Hello world
```

inside.

Probably you want to perform some actions depending on which stage you are, so that you can pass the stage name to the command by including a `%s` output conversion token in the command. For instance

```
hello "echo Hello world, stage %s"
```

Moreover, you can have also the time step currently executing and the current simulation time by including a `%d` and a `%f` output conversions, for instance

```
hello "echo Hello world, stage %s, step %d, time %f"
```

The order is important! That is, the first argument is the step (a C string), the second the time step (an integer) and the last the simulation time (a double).

In fact, basically, what PETSc-FEM does is to build string with `sprintf()` and then execute it with `system()` like

```
sprintf(command,your_command,stage,step,time);
system(command);
```

(see the Glibc manual for more info about sprintf and system). If you need, for some reason to switch the order then use a parameter number like

```
hello "echo Hello world, time %3$f, step %2$d, stage %1$s"
```

If you want to do some things depending on the stage then perhaps you can write something like this

```
hook_list shell_hook hello
hello "my_script_hook"
```

and

```
#!bin/bash
## File ‘my_script_hook’

if [ "$1" == "init" ]
then
    echo "in init"
    ## Do more things in ‘init’ stage
    ## ....
elif [ "$1" == "pre" ]
then
    echo "in pre"
    ## Do more things in ‘pre’ stage
    ## ....
elif [ "$1" == "post" ]
then
    echo "in post"
    ## Do more things in ‘post’ stage
    ## ....
elif [ "$1" == "close" ]
then
    echo "in close"
    ## Do more things in ‘close’ stage
    ## ....
else
    ## Catch all. Should not enter here.
    echo "Don’t know how to handle stage: $1"
fi
```

At the `init` and `close` hook-launching points the step number passed is -1 and -2 respectively, so that you can detect whether you are in a pre/post stage or init/close by checking this too. The time passed is in both cases 0.

10.4 Shell hooks with "make"

If no command is given, i.e. if you write

```
hook_list shell_hook hello
```

but don't add the

`hello` <*command*>

line, then PETSc-FEM uses a standard command line like this

```
make petscfem_step=%2$d petscfem_time=%3$f hello_%1$s
```

so that it will execute make commands with targets `hello_init`, `hello_pre`, `hello_post` and `hello_close` like

```
$ make petscfem_step=-1 petscfem_time=0. hello_init
$ make petscfem_step=1 petscfem_time=0.1 hello_pre
$ make petscfem_step=1 petscfem_time=0.1 hello_post
$ make petscfem_step=2 petscfem_time=0.2 hello_pre
$ make petscfem_step=2 petscfem_time=0.2 hello_post
$ make petscfem_step=3 petscfem_time=0.3 hello_pre
$ make petscfem_step=3 petscfem_time=0.3 hello_post
...
$ make petscfem_step=100 petscfem_time=10. hello_pre
$ make petscfem_step=100 petscfem_time=10. hello_post
$ make petscfem_step=-2 petscfem_time=0. hello_close
```

Inside the `Makefile` you can use the `make` variables `$(petscfem_step)` and `$(petscfem_time)`. For instance you can do the *"Hello world"* trick by adding the targets

```
# In Makefile
hello_init:
        echo "In init"
        ## Do more things in ‘init’ stage
        ## ....

hello_pre:
        echo "In pre"
        ## Do more things in ‘pre’ stage
        ## ....

hello_post:
        echo "In post"
        ## Do more things in ‘post’ stage
```

```
        ## ....

hello_close:
        echo "In close"
        ## Do more things in ‘close’ stage
        ## ....
```

For instance, I love to gzip my state files with a command like this

```
## In the PETSc-FEM data file
hook_list shell_hook compress

## In the Makefile
compress_init:
compress_pre:
compress_post:
        for f in *.state.tmp ; do echo "gzipping $f" ; gzip -f $f ; done
compress_close:
```

11 Gatherers and embedded gatherers

Typically *"gatherers"* are reduction operators over the element sets, i.e. instead of assembling a matrix or vector, the gather operations produce some kind of global data, as for instance

- The volume/surface/length of an elemset.
- The integral of some quantity (fields, coordinates, or functions of the them) over the element set, for instance total concentration, total energy, total (linear or angular) momentum or angular momentum.
- Same as above but depending also on gradients of the state variables, for instance: total elastic energy. This involves some
- Same as above but integrals of quantities on a manifold of a dimension lower than the embedding space. In this case the integrand may involve also the normal to the surface, for instance total flow of heat or volume rate through a surface.

Gatherers are implemented as `elemsets` the only difference is that they typically only process a special jobinfo named `gather`. This jobinfo is processed after the time step, i.e. only after the Newton loop, on the converged values. This jobinfo task does not assemble vectors or matrices, but a series global values stored in a global C++ vector called `vector<int> gather_values`. A typical call is as follows (taken from `ns.cpp`),

```
arglf.clear();
arglf.arg_add(&state,IN_VECTOR|USE_TIME_DATA);
arglf.arg_add(&state_old,IN_VECTOR|USE_TIME_DATA);
arglf.arg_add(&gather_values,VECTOR_ADD);
ierr = assemble(mesh,arglf,dofmap,"gather",&time_star);
CHKERRA(ierr);
```

Note that three arguments are passed to the `gather` assemble task: the old and new states, and the vector of gathered values.

Many gatherer elemsets have the suffix `integrator` appended to their names, for instance `visc_force_integrator` or `volume_integrator`

11.1 Dimensioning the values vector

In order to use the gatherers the user must before dimension appropriately the array of values with global option `ngather`. Then, for each gatherer elemset the user must set the options that select a continuous range in this vector, namely `gather_pos` and `gather_length`. The selected range is `[gather_pos,gather_pos+gather_length]`. Then, for instance, if the (hypothetical) gatherer elemset `momentum_integrator` is supposed to compute the integral of the momentum (a 3-vector in 3D), then we could use as

```
global_options
...
ngather 3
__END_HASH__

elemset momentum_integrator 4
...
gather_pos 0
gather_length 3
data ./connectivity.dat
__END_HASH
```

With this setup, the program will compute for each time step the integral of the momentum, at will print these three values on standard output. If a string is passed to the global option `gather_file` for instance

```
...
ngather 3
gather_file momentum.out
...
```

then, instead of reporting the gathered values on standard output they are printed on the corresponding file. The file is opened and closed at each time step.

11.2 Embedded gatherers

If the elemset has a dimension lower than the embedding space, for instance a surface embedded in 3D space, then computing the gradients of the variables on the surface can not be done with the information on the surface only. This is not an uncommon situation, for instance it happens when computing viscous forces of a Newtonian fluid on the skin of a solid body. The *gradient* of the velocity field must be computed in order to compute

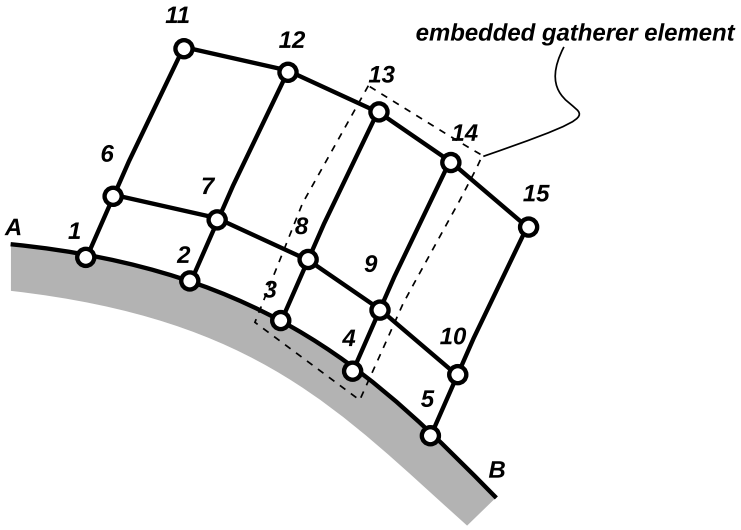


Figure 12: Embedded gatherer element

the stress on the skin. But if the velocity field is known only on the surface (think, for simplicity, in the case a of plane surface) the normal component of the gradient can not be determined.

In order to solve this, a special class of gatherers have been developed, namely the `embedded_gatherer` class. Typically such an elemset is composed is a surface elemset, associated with a volume one. For instance (see figure 12) the user can have a fluid problem with a volume elemset (in 2D) composed of `cartesian2d` and wants to compute the viscous traction on the solid surface AB. For this, she adds an elemset `visc_force_integrator` composed of six-nodes elements composed of three layers of segments parallel to the surface as, for instance, the element 3-4-8-9-14 (marked with a dashed line in the figure). This special elements can be seen as layers of surface elements, parallel to the skin. With the velocity values computed by the Navier-Stokes solver at these nodes, the gatherer elemset can compute high precision approximations to the normal derivatives of velocity at the surface.

- Note that this requires that the mesh must be somewhat structured near the surface. However, it is usual to add structured layer on the body skin in order to correctly capture the boundary layer.
- Also, it requires the construction of the connectivities of these layers, what may be cumbersome, but we will see later that this can be done automatically (see §11.3).
- The size of the elements in the normal direction are not required to be equispaced, i.e. the distances 1-6 and 6-11 are not required to be equal or similar. This is important, because normally the layers of nodes are refined towards the surface, as shown in the figure.
- The lines of nodes are not required to be normal to the surface.
- However, it is required that each row of nodes (for instance the row 1-6-10) must *lay on a smooth curve*. This is required, since in the process of computing high normal derivatives a Taylor expansion is computed in terms of this curves, so the error depends on the higher derivatives (e.g. the curvature) of the line.

The typical invocation is as follows

```
elemset nsi_tet_les_full 4
geometry cartesian2d
__END_HASH__
1 2 7 6
6 7 12 11
2 3 8 7
...
__END__ELEMSET__

elemset visc_force_integrator 6
geometry line2quad
__END_HASH__
1 2 6 7 11 12
2 3 7 8 12 13
3 4 8 9 13 14
...
__END__ELEMSET__
```

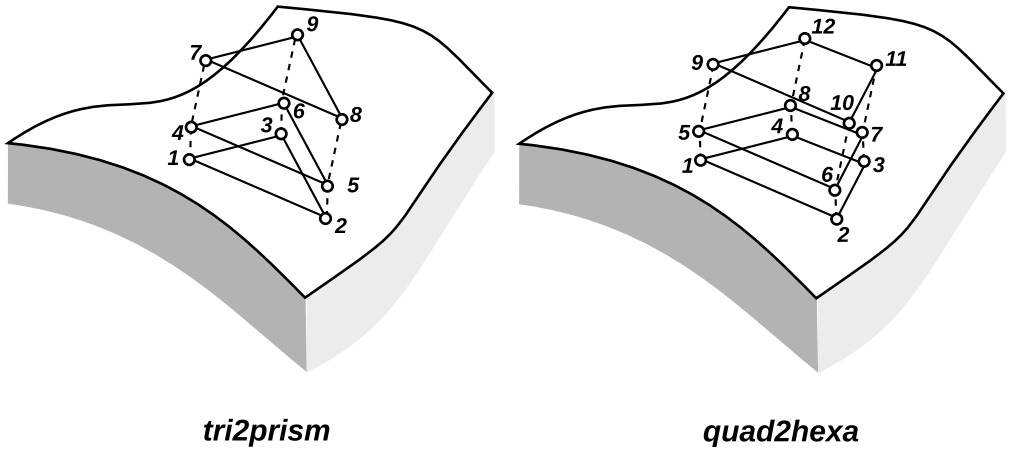


Figure 13: Embedded gatherer elements in 3D

Note that the geometry is special `line2quad` means that the surface geometry are lines and the corresponding volume elemset is composed of quads. The other two possibilities implemented so far are `tri2prism` and `quad2hexa` (see figure 13).

Typical invocation is as follows The typical invocation is as follows

```
elemset visc_force_integrator 9
geometry tri2prism
__END_HASH__
...
1 2 3 4 5 6 7 8 9
```

```
...
__END__ELEMSET__
```

and

```
elemset visc_force_integrator 12
geometry quad2hexa
__END_HASH__
...
1 2 3 4 5 6 7 8 9 10 11 12
...
__END__ELEMSET__
```

11.3 Automatic computation of layer connectivities

Sometimes it is somewhat cumbersome to compute the connectivities of the embedded gatherer connectivities since it involves the finding of layers of nodes inside the adjacent volume element. This can be done automatically by PETSc-FEM if the `identify_volume_elements` is activated and the name of the adjacent volume element is passed through the option `volume_elemset` for instance

```
elemset nsi_tet_les_full 4
geometry cartesian2d
name viscous_fluid
__END_HASH__
1 2 7 6
6 7 12 11
2 3 8 7
...
__END__ELEMSET__

elemset visc_force_integrator 6
volume_elemset viscous_fluid
identify_volume_elements
geometry line2quad
__END_HASH__
1 2 1 1 1 1
2 3 1 1 1 1
3 4 1 1 1 1
...
__END__ELEMSET__
```

Note that the nodes in the inner layers of nodes are replaced by 1's. With this setting, the code will inspect the connectivity of the `viscous_fluid` elemset and find the nodes corresponding to the inner layers and replace the 1's by the correct n ode number.

11.4 Passing element contributions as per-element properties

For some applications it is desirable to have the individual element contributions instead of having their sum. For instance, in a fluid-structure application involving a fluid and a deformable solid, it does not suffice only with the integral of the forces to compute the evolution of the solid, but also it is needed the whole distribution of forces. In this case what is needed is a list of surface elements and the total force for each element.

There are two flavors of this feature. The simplest one is activated with `dump_props_to_file` and simply stores the data in a file. The other possibility is activated with the `pass_values_as_props` option and stores the computed per-element values in the per-element properties table. Subsequent parts of the code can grab this information using the usual mechanism to query the per-element properties table.

All three mechanism can be activated or deactivated independently of the others. In summary

- Computation of global values is activated with `gather_length>0`.
- Storing the per-element is activated with the `dump_props_to_file` option.
- Passing the per-element computed values to other sections of the code via the per-element table is activated with the `pass_values_as_props` option.

In the last two cases the number of values to be computed by the gatherer can be set via the `store_values_length` option.

The summary of relevant options is

- `pass_values_as_props` activates the mechanism of passing per element computed values as per-element properties.
- `store_values_length` is an integer indicating how many values are computed by the gatherer. It defaults to `gather_length`. However, if the standard mechanism of the `gather_values` is deactivated, then the number of computed values must be passed through this option.
- `store_in_property_name` is a string that identifies the per-element property that must be filled with the computed values. Of course, the user must define such property with the appropriate size. The values in the connectivity table are irrelevant (for instance null values) and will be overwritten by the gatherer.
- `compute_densities`: If`compute_densities==0` then the value set in the per-element property is the integral of the integrand over the element. That is, if the integrand is ϕ, the computed value for the element is

$$\texttt{value for element e} = \int_{\Omega_e} \phi \tag{115}$$

  Conversely, if `compute_densities==1` the value set is the density of the mean value of the integrand, i.e.

$$\texttt{value for element e} = \langle \phi \rangle_{\Omega_e} = \frac{1}{|\Omega_e|} \int_{\Omega_e} \phi \tag{116}$$

  For instance, if the integrand ϕ is the heat flow through the surface, then if`compute_densities==0` then the value set in the per-element property is the total

heat flow through the element (which has units of energy per unit time), whereas if`compute_densities==1` then the value set is the mean heat flow density (which has units of energy per unit time and unit surface). For the traction on a surface, the passed value is the total force on the element in one case (units of force) and the mean skin friction in the other (units of force per unit area). Of course, this option has no effect on the values passed via the global `gather_values` vector.

- `dump_props_to_file` activates the mechanism of storing per-element computed values in a file.
- `dump_props_file` is the name of the file where the per-element values are written. If it contains a `%d` format sequence then it is replaced by the time step number (using `printf()` and friends) .
- `dump_props_freq` is the frequency at which the values are dumped to file.

11.5 Parallel aspects

Of course, PETSc-FEM is in charge of adding the contributions on different processors. The resulting sum of contributions over *all elements* in *all processors*. The sum is available not only in the master (`rank=0`), but in all processors (as with an `MPI_Allreduce()` call). This is important since this global sum can be used also in *hooks* in order to perform computations. For instance, a gatherer can be used to compute the force on a body, and this force can be passed to a hook to compute the movement of the body.

11.6 Creating a gatherer

Typically a gatherer is created by deriving from the virtual class `gatherer` and implementing the `set_pg_values()` method which is in charge of computing the integrands.

```
class gatherer {
// ...
public:
  // perform several checks and initialization
  void init();
  // add Gauss point contributions
  void set_pg_values(vector<double> &pg_values,FastMat2 &u,
     FastMat2 &uold,FastMat2 &xpg,FastMat2 &Jaco,
     double wpgdet,double time);
};
```

- The `init()` function may be used for initialization. The `TGETOPTDEF(thash,....)` macro can be used for extracting options of the elemset. For instance,

  ```
  TGETOPTDEF(thash,double,Young_modulus,0.);
  ```

- The `set_pg_values(...)` is the main method. Here the user computes the values to be integrated. Its signature is

```
void set_pg_values(vector<double> &pg_values,FastMat2 &u,
   FastMat2 &uold,FastMat2 &xpg,FastMat2 &n,
   double wpgdet,double time);
```

The argument `pg_values` are the values to be computed. (`pg` stands for Gauss point, since usually this function is called by the gatherer class at the Gauss points of integration.) `u` and `uold` are the states at times t^n and t^{n+1}. `xpg` are the coordinates of the Gauss point. `n` is the normal to the surface (only relevant if the dimension of the element `ndimel` is equal to `ndim-1`). `wpgdet` is the area of the Gauss point (i.e. the Gauss point weight times the Jacobian of the transformation to the master element).

- `void element_hook(int k)` is called *before* the Gauss point hook and then can be used in order to pre-compute some stuff for all the Gauss points. `k` is the element number.
- `void clean()` can be used after processing all the elements and doing some cleanup.

12 Generic load elemsets

Generic load elemsets account for surface contributions which represent constant terms in the governing equations or either a function of the state at the surface. Typical terms that can be represented in this way are

- External heat loads (like a constant radiation load) in thermal problems: $q = \bar{q}$
- A linear Newtonian cooling term $q = -h(T - T_\infty)$.
- A nonlinear Newtonian cooling term $q = f(T, T_\infty)$.

12.1 Linear generic load elemset

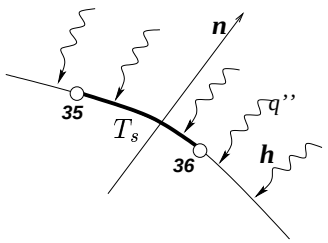


Figure 14: Generic load element (single layer)

In the simplest case the load is of the form

$$\begin{aligned} \mathbf{q} &= -\mathbf{h}\mathbf{U} + \bar{\mathbf{q}}, && \text{(single layer)} \\ \mathbf{q} &= \mathbf{h}(\mathbf{U}_{\text{out}} - \mathbf{U}) + \bar{\mathbf{q}}, && \text{(double layer)} \end{aligned} \tag{117}$$

This is implemented by the `lin_gen_load` elemset. This elemset may be *"single-layer"* or *"double-layer"* (see figures 14 and 15). Double layer elements can represent a lumped thermal resistance, for instance a very thin layer of air inside a material of higher conductivity. In the double layer case the number of nodes is twice than in the single-layer

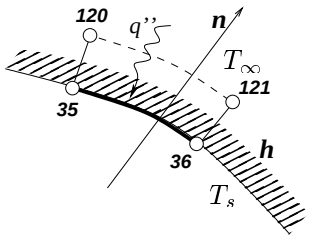


Figure 15: Generic load element (double layer)

case. Double layer is activated either by including the `double_layer` option or either if the numer of nodesis twice that one specified by the geometry.

The options for the `lin_gen_load` elemset are

- `int double_layer` (default=0):

  Whether there is a double or single layer of nodes (found in file: `genload.cpp`)

- `string geometry` (default=`cartesian1d`):

  Type of element geometry to define Gauss Point data (found in file: `genload.cpp`)

- `double[var_len] hfilm_coeff` (default= `no default`):

  Defines coeffcients for the film flux function. May be `var_len=0` (no ΔT driven load) or `var_len=ndof*ndof` a full matrix of relating the flux with $\Delta \mathbf{U}$. (found in file: `linhff.cpp`)

- `double[var_len] hfilm_source` (default= `no default`):

  Defines constant source term for the generic load on surfaces. May be of length 0 (null load) or `ndof` which represents a geven load per field. (found in file: `linhff.cpp`)

- `int ndimel` (default=`ndim-1`):

  The dimension of the element (found in file: `genload.cpp`)

12.2 Functional extensions of the elemset

The generic elemset is `GenLoad`. There is an instantiation for the Generic load elemsets can be extended by the user by deriving the base class `GenLoad` and the most important task is to implement the methos `q(u,flux,jac)` for the single layer case, and `q(u_in.u_out,flux_in,flux_out,jac)` in the double layer case.

12.3 The flow reversal elemset

This instantiation of `GenLoad` is useful for avoiding instabilities caused by infersion of the flow at an outlet boundary. Assume that the Navier-Stokes module computes a velocity field $\mathbf{v}$ and some scalar ϕ is being advected with this velocity field. On an outlet boundary one usually leaves the temperature free, that is, no Dirichlet condition is imposed on ϕ. But if the flow is reversed, that is $\mathbf{v} \cdot \hat{\mathbf{n}} < 0$ ($\hat{\mathbf{n}}$ is the external normal to the boundary) at some instant, then this boundary condition becomes ill-posed and the simulation may diverge.

One solution is to switch from Neumann to Dirichlet boundary conditions depending on the sign of $\mathbf{v} \cdot \hat{\mathbf{n}}$, i.e.

$$\mathbf{q} = \begin{cases} 0, & \text{if } \mathbf{v} \cdot \hat{\mathbf{n}} \geq 0, \\ h(\bar{\phi} - \phi), & \text{if } \mathbf{v} \cdot \hat{\mathbf{n}} > 0, \end{cases} \tag{118}$$

where h is a film coefficient and $\bar{p}hi$ is the value to be imposed at the boundary when the flow is reversed. Note that this amounts to switch to a Dirichlet condition by *penalization*. For h large enough, the value of ϕ will converge to $\bar{\phi}$. However, if h is too large, this can affect the conditioning of the linear system.

The options for this elemset are

- `vector<double> coefs` (default= `empty vector`): Penalization coefficients (artificial film coefficients). The length of `coefs` and `dofs` must be the same. (found in file: `flowrev.cpp`)
- `vector<int> dofs` (default= `empty vector`): Field indexes (base-1) for which the flow reversal will be applied (found in file: `flowrev.cpp`)
- `int ndim` (default=0):

  Dimension of the problem (found in file: `flowrev.cpp`)
- `vector<double> refvals` (default= `empty vector`): The reference values for the unknown. The length of `coefs` and `dofs` must be the same. (found in file: `flowrev.cpp`)
- `int thermal_convection` (default=0):

  If this is set, then the code assumes that a thermal NS problem is being run i.e. `vel_index=1` and the penalization term is added only on the temperature field (i.e. dofs=[ndim+2]). Also the refval (found in file: `flowrev.cpp`)
- `int vel_indx` (default=1):

  Field index where the components of the velocity index start (1-base). (found in file: `flowrev.cpp`)

13 Visualization with DX

Data Explorer (http://www.opendx.org) is a system of tools and user interfaces for visualizing scientific data. Originally an IBM commercial product, has been released now under the IBM Open Source License, and maintained by a group of volunteers (see the URL above). Besides their impressive visualization capabilities, DX has many features that make it an ideal visualization tool for PETSc-FEM.

- DX is Open Source with a License very close to the GPL (not completely compatible though)
- It has a *"Visual Program Editor"* which makes it very configurable for different modules.

- It has been linked to PETSc-FEM through sockets, which makes it possible to visualize a running job, even in background.
- It has a scripting language.

If you want to visualize your results with DX you have first to download it from the URL above and install it. Then you can pass your results to DX simply by editing the needed `.dx` files or well by using the `ExtProgImport` DX module. In order to use this last option you have to

- Compile PETSc-FEM with the `USE_SSL` flag enabled (disabled by default). Also, if you want to DX be able to communicate asynchronously with PETSc-FEM, you have to compile with the `USE_PTHREADS` flag enabled (disabled by default)
- Load the `dx_hook` hook in PETSc-FEM and pass it some options.
- Build the dynamically loadable module (file `dx/epimport`) and load it in DX.

DX basic visualization units are `Field` objects, which are composed of three `Array` objects called `positions` (node coordinates), `connections` (element connectivities) and `data` (computed field values). At each time step, `ExtProgImport` exports two `Group` objects,

- a `Group` of `Arrays` named `output_array_list` and
- a `Group` of `Fields` objects named `output_fields_list`.

This objects are generated as follows.

- For each `Nodedata` PETSc-FEM object a `positions` array is constructed, this results in, say, `nn` array objects. (Currently PETSc-FEM has only one `Nodedata` object (member name `nodes`), so that `NN=1`).
- A `data` array is constructed in basis to the current state vector (member name `data`).
- For each elemset, a `connections` array is constructed. You can disable construction of some particular elemsets through the `dx` elemset option, and also which nodes and DX interpolation geometry is used. This results in other `ne` connection arrays. The member name of each array is based in the `name` option of the elemset. If this has not been set, the name is set to the type of the elemset. If collision is produced, a suffix of the form `_0`, `_1` is appended, in order to make it unique. Options controlling how the connection array is constructed can be consulted in §4.4.2.

The resulting `nn+ne+1` arrays are grouped in a `Group` object and sent through the `output_array_list` output tab. You can extract the individual components with the `Select` DX module, and build field objects. Also a set of field objects is created automatically and sent through the `output_field_list` output tab. Basically a field is constructed for each possible combination of positions, connections and data objects. This may seem a huge amount of fields, but in fact as the arrays are passed internally by pointers in DX,

the additional memory requirements are not large. (At the time of writing this, this is a set of `nn*ne=ne` fields, since `nn=1`). A name is generated automatically for each field.

Some information is send to the DX *"Message Window"*. Also it's very useful to put `Print` modules downstream of the `ExtProgImport` module in order to see which arrays and fields have been created.

The communication between DX and PETSc-FEM is done through a socket. PETSc-FEM acts as a *"server"* whereas DX acts as a *"client"*. PETSc-FEM opens a *"port"* (option `dx_port`), and DX connects to that port (the *"port"* input tab in the `ExtProgImport` module). (Currently, the standard port for the DX/PETSc-FEM communication is 5314.) DX can communicate with PETSc-FEM running in the background ad even on other machine (the *"serverhost"* input tab). At each time step, DX sends a request to PETSc-FEM which answers sending back *"arrays"* and *"fields"*.

13.1 Asynchronous/synchronous communication

- In *"synchronous"* mode (`steps>0`) PETSc-FEM waits each `steps` time steps a connection from DX. Once DX connects to the port, PETSc-FEM transmits the required data and resumes computation. This is the appropriate way of communication when generating a sequence of frames for a video with a DX sequencer, for instance. Note that if you don't use a sequencer then you have to arrange in someway to make `ExtProgImport` awake and connect to PETSc-FEM, otherwise the update in the visualization is not performed and the PETSc-FEM job is stopped, waiting for the connection.

- In *"asynchronous"* mode (`steps=0`), in contrast, PETSc-FEM monitors each port after computing a time step. If a DX client is trying to connect, it answers the request and resumes computing, otherwise it resumes computing immediately. This is ideal for monitor a job that is running in background, for instance. Note that in this case the interference with the PETSc-FEM job is minimal, since once PETSc-FEM answers the request, resumes processing automatically until a new connection is requested.

The `steps` state variable is internal to PETSc-FEM. It can be set initially with a `dx_steps` options line (1 by default). After that, it can be changed by changing the `steps` input tab. However, note that the change doesn't take effect until the next connection of DX to PETSc-FEM. If you don't want to change the internal state of the `steps` variable then you can set it to `NULL` or `-1`.

13.2 Building and loading the ExtProgImport module

This module allows PETSc-FEM to exchange data with DX through a socket, using a predefined protocol. The module is in the `$(PETSCFEM_DIR)/dx` directory of the PETSc-FEM distribution. To built it, you have to compile first the `petscfem` library, and then `cd` to the `dx` directory and make `$ make`. This should build the `dx/epimport` file, which is a dynamically loadable module for DX. This one altogether with the `dx/epimport.mdf` (which is a plain text file describing the inputs/outputs of the module, and other things) are the files needed by DX in order to run the module.

To load this module in DX you can do this either at the moment of launching DX with something like

```
$ dx -mdf /path/to/epimport.mdf
```

or well from the `dxexec` window (menu `File/Load Module Descriptions`).

13.3 Inputs/outputs of the ExtProgImport module

- (input) `steps`: type `integer`; default `0`. *Description:* Visualize each "steps" time steps. (0 means asynchronously).
- (input) `serverhost`; type `string`; default `"localhost"`; *Description:* Name of host where the external program is running. May be also set in dot notation (e.g. `200.9.223.34` or `myhost.mydomain.edu`).
- (input) `port`; type `integer`; default `5314`; *Description:*Port number
- (input) `options`; type `string`; default `NULL`; *Description:* Options to be passed to the external program
- (input) `step`; type `integer`; default `-1`; *Description:* Input for sequencer. This value is passed to the `dx_hook` hook, but currently is ignored by it. It could be used in a future in order to synchronize the DX internal step number with the PETSc-FEM one.
- (input) `state_file`; type `string`; default `NULL`; An ASCII file storing a state where to read the state to be visualized.
- (output) `output_array_list`; type `field`; *Description:* Group object of imported arrays
- (output) `output_field_list`; type `field`; *Description:* Group object of imported fields

13.4 DX hook options

- `int dx_auto_combine` (default=`0`):

  Auto generate states by combining elemsets with fields (found in file: `dxhook.cpp`)

- `int dx_cache_coords` (default=`0`):

  Uses DX cache if possible in order to avoid sending the coordinates each time step or frame. Use only if coordinates are not changing in your problem. (found in file: `dxhook.cpp`)

- `double dx_coords_scale_factor` (default=`1.`):

  Coefficient affecting the new displacements read. New coordinates are `c0*x0+c*u` where `c0=dx_coords_scale_factor0` , `c1=dx_coords_scale_factor` , `x0` are the initial coordinates and `u` are the coordinates read from the given file. (found in file: `dxhook.cpp`)

- `double dx_coords_scale_factor0` (default=`1.`):

  Coefficient affecting the original coordinates. See doc for `dx_coords_scale_factor` (found in file: `dxhook.cpp`)

- `int dx_do_make_command` (default=`0`):

  If true, then issue a `make dx_step=<step> dx_make_command` (found in file: `dxhook.cpp`)

- `string dx_node_coordinates` (default=`<none>`):

  Mesh where updated coordinates must be read (found in file: `dxhook.cpp`)

- `int dx_port` (default=`5314`):

  TCP/IP port for communicating with DX (`5000 < dx_port < 65536`). (found in file: `dxhook.cpp`)

- `int dx_read_state_from_file` (default=`0`):

  Read states from file instead of computing them . Normally this is done to analyze a previous run. If 1 the file is ASCII, if 2 then it is a binary file. In both cases the order of the elements must be: `u(1,1)`, `u(1,2)`, `u(1,3)`, `u(1,ndof)`, `u(2,1)`, ... `u(nnod,ndof)` where `u(i,j)` is the value of field `j` at node `i`. (found in file: `dxhook.cpp`)

- `string dx_split_state` (default=):

  Generates DX fields by combination of the input fields (found in file: `dxhook.cpp`)

- `int dx_state_all_fields` (default=`!dx_split_state_flag`):

  Generates a DX field with the whole state (all ndof fields) (found in file: `dxhook.cpp`)

- `int dx_steps` (default=`1`):

  Initial value for the `steps` parameter. (found in file: `dxhook.cpp`)

- `int dx_stop_on_bad_file` (default=`0`):

  If `dx_read_state_from_file` is activated and can't read from the given file, then stop. Otherwise continue padding with zeros. (found in file: `dxhook.cpp`)

14 The "idmap" class

14.1 Permutation matrices

This class represents sparse matrices that are "close" to a permutation. Assume that the $n \times n$ matrix Q is a permutation matrix, so that if $\mathbf{y}$, $\mathbf{x}$ are vectors of dimension n and

$$\mathbf{y} = \mathbf{Q}\,\mathbf{x} \tag{119}$$

then

$$y_j = x_{P(j)} \tag{120}$$

where P is the permutation associated with Q. For instance if $P = \{1, 2, 3, 4\} \rightarrow \{2, 4, 3, 1\}$ then

$$\mathbf{y} = \mathbf{Q}\,\mathbf{x} = \begin{bmatrix} x_2 \\ x_4 \\ x_3 \\ x_1 \end{bmatrix} \tag{121}$$

This matrix can be efficiently stored as an array of integers of dimension ident[n] such that ident[j − 1] = $P(j)$. Also, the inverse permutation can be stored in another integer array iident[n], so that both Q and Q^{-1} can be stored at the cost of n integer values for each. Also, both multiplication and inversion operations as $\mathbf{y} = \mathbf{Q}\,\mathbf{x}$, $\mathbf{y} = \mathbf{Q}^T\,\mathbf{x}$, $\mathbf{x} = \mathbf{Q}^{-1}\,\mathbf{y}$ and $\mathbf{x} = \mathbf{Q}^{-T}\,\mathbf{y}$, can be done with n-addressing operations.

14.2 Permutation matrices in the FEM context

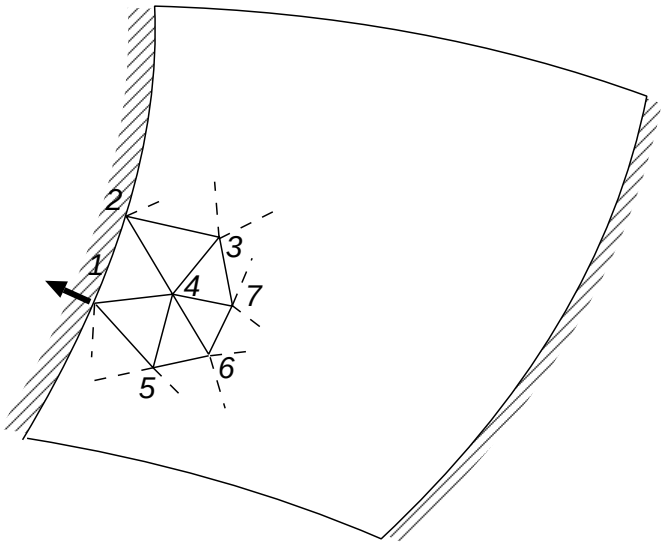


Figure 16: Node/field to dof map. Example for NS or Euler in a duct.

Permutation matrices are very common in FEM applications for describing the relation between node/field pairs and degrees of freedom (abbrev. *"dof"*'s). For instance consider a flow NS or Euler application in a mesh like that shown in figure 16. At each node we have a set of unknown "fields" (both components of velocity and pressure, u, v and p). In

a first description, we may arrange the vector of unknowns as

$$\mathbf{U}_{\mathrm{NF}} = \begin{bmatrix} u_1 \\ v_1 \\ p_1 \\ u_2 \\ v_2 \\ p_2 \\ \vdots \\ u_{N_{\mathrm{nod}}} \\ v_{N_{\mathrm{nod}}} \\ p_{N_{\mathrm{nod}}} \end{bmatrix} \tag{122}$$

The length of this vector is $N_{\mathrm{nod}} \times n_{\mathrm{dof}}$ and this may be called the "node/field" (this accounts for the NF subindex) description of the vector of unknowns. However, we can not take this vector as the vector of unknowns for actual computations due to a series of facts,

- Not all of them are true unknowns, since some of them may be imposed to a given value by a Dirichlet boundary condition.
- There may be some constraints between them, for instance, in structural mechanics two material points linked by a rigid bar, or a node that is constrained to move on a surface or line. In CFD these constraints arise when periodic boundary conditions are considered
- Some reordering may be necessary, either for reducing band-width if a direct solver is used, or either due to mesh partitioning, if the problem is solved in parallel.
- Also, non-linear constraints may arise, for instance when considering absorbing boundary conditions or non-linear restrictions.

So that we assume that we have a relation

$$\mathbf{U}_{\mathrm{NF}} = \mathbf{Q}\,\mathbf{U} + \bar{\mathbf{Q}}\,\bar{\mathbf{U}} \tag{123}$$

where $\mathbf{U}$ is the "reduced" array of unknowns, $\bar{\mathbf{U}}$ representing the externally fixed values, and Q, Q^* appropriated matrices. Follows some common cases

Dirichlet boundary conditions: If the velocity components of node 1 are fixed, then we have

$$\begin{aligned} u_1 &= \bar{u}_1 \\ v_1 &= \bar{v}_1 \end{aligned} \tag{124}$$

so that the corresponding file in $\mathbf{Q}$ is null and the file in $\bar{\mathbf{Q}}$ has a "1" in the entry corresponding to the barred values.

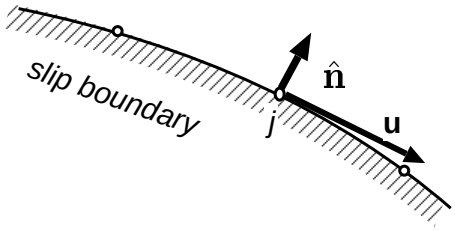


Figure 17: Slip boundary condition

Slip boundary conditions: For the Euler equations, if node j is on a slip wall and $\hat{\mathbf{n}}_j$ is the corresponding normal, then the normal component of velocity is null and the tangential component is free. (In 3D there are two free tangential components and one prescribed normal component). The normal component may be prescribed to some non-null value if transpirations fluxes are to be imposed. The corresponding equations may look like this

$$\begin{aligned} u_j &= u_{jt} t_x + \bar{u}_{jn} n_x \\ v_j &= u_{jt} t_y + \bar{u}_{jn} n_y \end{aligned} \tag{125}$$

where u_{jt} is the (free) tangential component and $\bar{u}_{jn}$ the prescribed normal component.

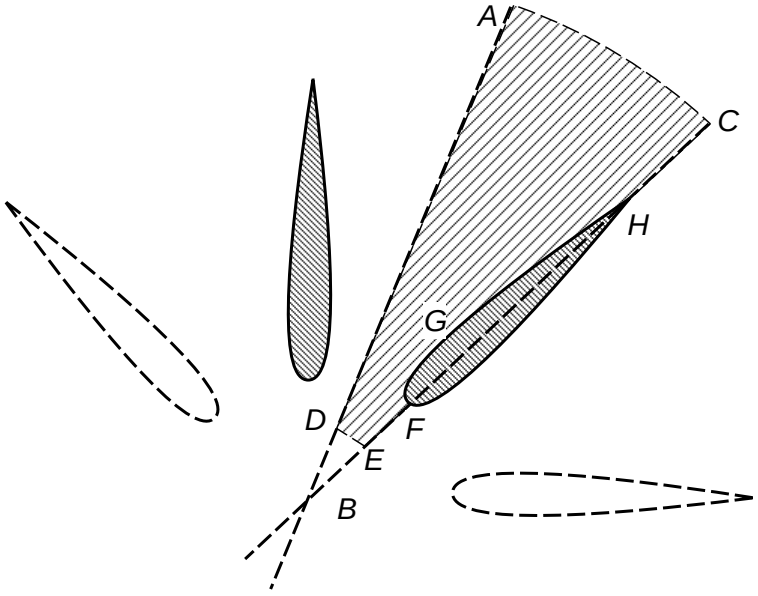


Figure 18: Symmetrical arrangement of foils with specular symmetry.

Periodic boundary conditions: These kind of b.c.'s are useful when treating geometries with repetitive components, as is very common for rotating machinery. In some cases this can be done as slip boundary conditions (see figure 18). Here the foils are symmetric about their centerline, and then the whole geometry not only posses

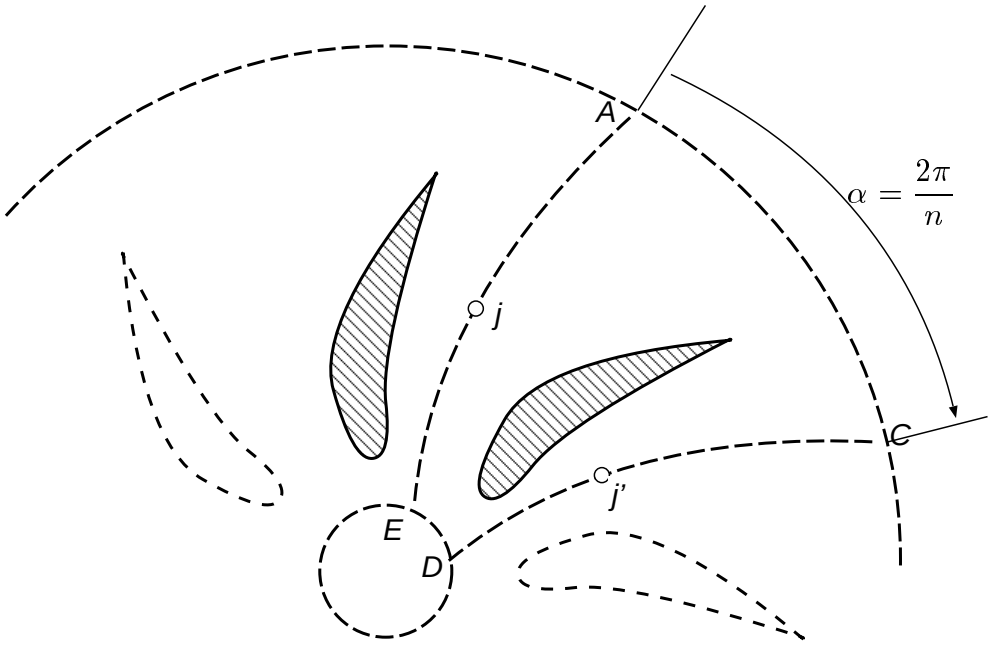


Figure 19: Non-symmetrical arrangement of foils.

symmetry of rotation about angles multiple of $2\pi/n_{\text{foils}}$ but in addition possess reflection symmetry about the centerline of a foil, like BC and the centerline between two consecutive foils like AB. In consequence it suffices the domain $ADEC$ with inlet/outlet b.c.'s in DE and AC, slip boundary conditions on those parts of AD and $EFGHC$. On the foil (boundary FGH) pressure should be free, while on AD, EF and HC we can impose $(\partial p/\partial n) = 0$. For Navier-Stokes we may impose solid wall b.c.'s on the foil, normal null velocity component on AD, EF and HC, null tangential shear stress $((\partial u_t/\partial n) = 0)$, and $(\partial p/\partial n) = 0$.

However, if the foils are non-symmetric or they are not disposed symmetrically about a line passing by the rotation axis (see figure 19) then there are no predefined streamlines, but given two corresponding point like j and j' that are obtained through rotation of $\alpha = 2\pi/n_{\text{foils}}$, then we can impose

$$\begin{aligned} u_{j'} &= \cos\alpha\, u_j + \sin\alpha\, v_j \\ v_{j'} &= -\sin\alpha\, u_j + \cos\alpha\, v_j \\ p_{j'} &= p_j \end{aligned} \tag{126}$$

Absorbing boundary conditions The basic concept about these b.c.'s is to impose the ingoing components from outside (reference) values while leaving the outgoing components free. If $\mathbf{w} = [u, v, p]$ is the state vector of the fluid at some node j on the outlet boundary (see figure 16), then $\mathbf{V}\mathbf{w} = \mathbf{u}$ are the eigen-components, where $\mathbf{V}$ is the change of basis matrix. The absorbing b.c. may be written as

$$\mathbf{u} = \mathbf{V} \begin{bmatrix} \bar{w}^1 \\ w^2 \\ w^3 \end{bmatrix} \tag{127}$$

Where $\bar{w}^1$ is the in-going component (taken from the reference value $\mathbf{w}_{\text{ref}}$) and the other components $w^{2,3}$ are free, i.e. they go in $\mathbf{U}$.

Non-linear Dirichlet boundary conditions: In some cases, Dirichlet boundary conditions are not expressed in terms of the state variables used in the computations, but on a non-linear combination of them, instead. For instance, consider the transport of moisture and heat through a porous media, and choose temperature T and moisture content H as the state variables. On an external boundary, we impose that the partial pressure of water should be equal to its external value $P_w = P_{w,\text{atm}}$. As the partial pressure of water (which may be related to relative humidity) is a complex non-linear function of T and H through the sorption isotherms of the porous media and the saturation pressure of water, it results in a non-linear link of the form $P_w(T, H) = P_{w,\text{atm}}$. By the moment we consider only a linear relation, since the non-linear case doesn't fit in the representation (123). The non linear case will be considered later on.

14.3 A small example

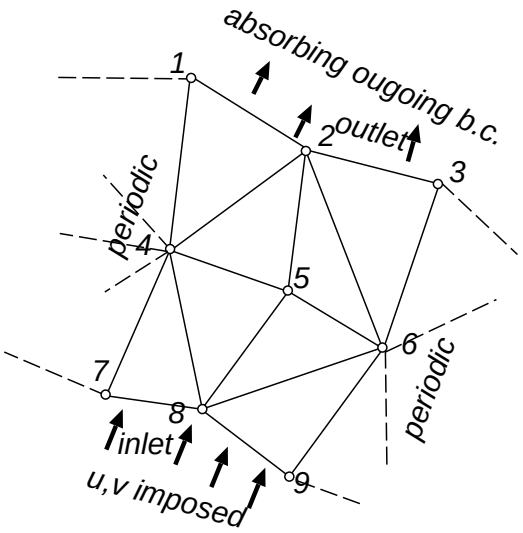


Figure 20: A small example showing boundary conditions.

Consider the region shown in figure 20, composed of 8 triangle elements and 9 nodes. We have $9 \times 3 = 27$ node/field values, but periodic boundary conditions on side 1-4-7 to side 3-6-9 eliminates the 9 unknowns on these last three nodes. In addition, at the outlet boundary (nodes 1-2-3) there is only one in-going component so that the unknowns here are only w_1^2, w_1^3, w_2^2 and w_2^3. On the other hand, on the inlet boundary u, and v are

imposed so the vector of unknowns is

$$\begin{aligned} U_1 &= w_1^2 \\ U_2 &= w_1^3 \\ U_3 &= w_2^2 \\ U_4 &= w_2^3 \\ U_5 &= u_4 \\ U_6 &= v_4 \\ U_7 &= p_4 \\ U_8 &= u_5 \\ U_9 &= v_5 \\ U_{10} &= p_5 \\ U_{11} &= p_7 \\ U_{12} &= p_8 \end{aligned} \tag{128}$$

while the prescribed values are

$$\begin{aligned} \bar{U}_1 &= \bar{w}_1^1 \\ \bar{U}_2 &= \bar{w}_2^1 \\ \bar{U}_3 &= u_7 \\ \bar{U}_4 &= v_7 \\ \bar{U}_5 &= u_8 \\ \bar{U}_6 &= v_8 \end{aligned} \tag{129}$$

And the relation defining matrices $\mathbf{Q}$ and $\bar{\mathbf{Q}}$ are

$$
\begin{aligned}
u_1 &= V_{11}^1 \bar{U}_1 + V_{12}^1 U_1 + V_{13}^1 U_2 \\
v_1 &= V_{21}^1 \bar{U}_1 + V_{22}^1 U_1 + V_{23}^1 U_2 \\
p_1 &= V_{21}^1 \bar{U}_1 + V_{22}^1 U_1 + V_{23}^1 U_2 \\
u_2 &= V_{11}^2 \bar{U}_2 + V_{12}^2 U_3 + V_{13}^2 U_4 \\
v_2 &= V_{21}^2 \bar{U}_2 + V_{22}^2 U_3 + V_{23}^2 U_4 \\
p_2 &= V_{21}^2 \bar{U}_2 + V_{22}^2 U_3 + V_{23}^2 U_4 \\
u_3 &= \cos\alpha\,(V_{11}^1 \bar{U}_1 + V_{12}^1 U_1 + V_{13}^1 U_2) + \sin\alpha\,(V_{21}^1 \bar{U}_1 + V_{22}^1 U_1 + V_{23}^1 U_2) \\
v_3 &= -\sin\alpha\,(V_{11}^1 \bar{U}_1 + V_{12}^1 U_1 + V_{13}^1 U_2) + \cos\alpha\,(V_{21}^1 \bar{U}_1 + V_{22}^1 U_1 + V_{23}^1 U_2) \\
p_3 &= V_{21}^1 \bar{U}_1 + V_{22}^1 U_1 + V_{23}^1 U_2 \\
u_4 &= U_5 \\
v_4 &= U_6 \\
p_4 &= U_7 \\
u_5 &= U_8 \\
v_5 &= U_9 \\
p_5 &= U_1 0 \\
u_6 &= \cos\alpha\, U_5 + \sin\alpha\, U_6 \\
v_6 &= -\sin\alpha\, U_5 + \cos\alpha\, U_6 \\
p_6 &= U_7 \\
u_7 &= \bar{U}_3 \\
v_7 &= \bar{U}_4 \\
p_7 &= U_{11} \\
u_8 &= \bar{U}_5 \\
v_8 &= \bar{U}_6 \\
p_8 &= U_{12} \\
u_9 &= \bar{U}_3 \\
v_9 &= \bar{U}_4 \\
p_9 &= U_{11}
\end{aligned}
\tag{130}
$$

As we can see the boundary conditions result in sparse $\mathbf{Q}$ and $\bar{\mathbf{Q}}$ matrices. Moreover, in real (large) problems most of the rows correspond to interior nodes (such as node 5 here) so that $\mathbf{Q}$ is very close to a permutation matrix. If we think at accessing the elements of $\mathbf{Q}$ by row, then could store $\mathbf{Q}$ as a sparse matrix, but in this case we need an average of two integers (for pointers) and a double (for the value) per unknown, whereas a permutation matrix can be stored with only one integer per unknown. One possibility is to think at these matrices as permutations followed by a sequence of operations depending on each kind of boundary conditions, but it may happen also that several kind of b.c.'s superposes, as int the case of node 3 (periodic + absorbing) and node 9 (periodic+ Dirichlet).

14.4 Inversion of the map

It is necessary sometimes to invert relation (123), i.e. given $\mathbf{U}_{\rm NF}$ and $\bar{\mathbf{U}}$ to find $\mathbf{U}$, for instance when initializing a temporal problem. Of course, this may not be possible for arbitrary $\mathbf{U}_{\rm NF}$ and $\bar{\mathbf{U}}$, since $\mathbf{Q}$ is in general rectangular, but we assume that $\mathbf{Q}^T\mathbf{Q}$ is non singular and solve (123) in a least square sense. After, we may verify if the given data ($\mathbf{U}_{\rm NF}$ and $\bar{\mathbf{U}}$) is *"compatible"* by evaluating the residual of the equation. This operation should be performed in $O(N)$ operations.

14.5 Design and efficiency restrictions

The class of matrices representing $\mathbf{Q}$ and $\bar{\mathbf{Q}}$ should have the following characteristics.

- Be capable of storing arbitrary matrices.
- Should be efficient for permutation like matrices, i.e. require as storage (order of) 2 integers per unknown and constant time access by row and column.

14.6 Implementation

An arbitrary permutation P of N objects can be stored as two integer vectors `ident`$[N]$ and `iident`$[N]$ such that if $P(j) = k$, then

$$\begin{aligned} \texttt{ident}[j-1] &= k \\ \texttt{iident}[k-1] &= j \end{aligned} \tag{131}$$

We will consider a slight modification to this. A row is

void: If all its elements are null

normal: If one element is one and the rest null (like in a permutation matrix).

special: otherwise.

Then we set

$$\texttt{ident}[j-1] = \begin{cases} 0; & \text{if row } j \text{ is void} \\ k; & \text{if row } j \text{ is normal and the non-null element is in position } k \\ -1; & \text{if row } j \text{ is special} \end{cases} \tag{132}$$

We need now how to access the coefficients of the special rows. Rows are stored as a "map", in the STL language, from integer values to doubles, i.e.

```
typedef map<int,double> row_t;
```

and we keep in class `idmap` a map from row indices to pointers to rows of the form

```
map<int,row_t*> row_map;
```

So that, for a given j we can get the corresponding row by checking the value of `inode`$[j-1]$. If the row is void or normal we return the corresponding row and if it is special, then we look for the pointer in `row_map` and this returns the desired row. Similar information is stored by columns, but in this case it is not necessary to store the values for the special rows so that the columns are of the form

```
typedef set<int> col_t;
```

and class `idmap` contains a private member `col_map` of the form:

```
map<int,col_t*> col_map;
```

14.7 Block matrices

When solving a linear (123) for $\mathbf{U}$, it is a key point to take into account that most part of $\mathbf{Q}$ are independent blocks, so that the inversion may be done with a minimum computational effort. We say that a given row and column indices i, j we say that they are *"directly connected"* if the element Q_{ij} is non null. We can then define that two row indices i, i' are *"indirectly connected"* if they are connected by a path $i \to j_1 \to i_1 \to j_2 \to \ldots \to j_n \to i'$ where the arrow means "directly connected". The definition applies also to pairs of column indices and row/column, column/row pairs. The indirect connection defines an equivalence class that splits the rows in disjoint sets.

14.7.1 Example:

Consider matrix

$$\mathbf{Q} = \left[\begin{array}{rcccccccccc} \text{row} \to & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 \\ \text{col} \downarrow 1 & . & . & . & . & . & . & . & * & * & . \\ 2 & . & . & . & . & . & . & . & . & . & . \\ 3 & . & . & . & . & . & . & . & * & * & . \\ 4 & . & . & . & . & . & . & . & . & . & * \\ 5 & . & . & . & * & * & . & * & . & . & . \\ 6 & . & . & . & * & * & . & * & . & . & . \\ 7 & * & . & . & . & . & . & . & . & . & . \\ 8 & . & . & . & . & . & * & . & . & . & . \\ 9 & . & . & . & . & . & . & . & . & . & . \\ 10 & . & . & . & . & . & . & . & * & * & . \end{array}\right] \tag{133}$$

where the asterisks means a non-null element. Starting with row index 1, we see that its corresponding connecting set is rows 1,3,10 and columns 8,9. So that the linear transfor-

mation $\mathbf{y} = \mathbf{Q}\mathbf{x}$ can be represent as

$$\begin{aligned} y_2 &= 0 \\ y_9 &= 0 \\ \begin{bmatrix} y_1 \\ y_3 \\ y_{10} \end{bmatrix} &= \mathbf{Q}^1 \begin{bmatrix} x_8 \\ x_9 \end{bmatrix} \\ y_4 &= Q^{4,10} x_{10} \\ \begin{bmatrix} y_5 \\ y_6 \end{bmatrix} &= \mathbf{Q}^3 \begin{bmatrix} x_4 \\ x_5 \\ x_7 \end{bmatrix} \end{aligned} \tag{134}$$

So that $\mathbf{Q}$ decomposes in three blocks of 3×2, 1×1, and 2×3, two void rows (2,9) and two row columns (2,3). One basic operation of class idmap is to compute the row connected to a given column index.

14.8 Temporal dependent boundary conditions

Temporal dependent boundary conditions are treated in PETSc-FEM assuming that the boundary condition on a set of nodes $J = \{j_1, j_2, j_3, \ldots, j_N\}$ is of the form

$$\begin{aligned} \phi_{j_1}(t) &= a_1 \phi(t) \\ \phi_{j_2}(t) &= a_2 \phi(t) \\ &\vdots \\ \phi_{j_N}(t) &= a_N \phi(t) \end{aligned} \tag{135}$$

where the a_k are spatial amplitudes and the function $\phi(t)$ is the temporal part of the dependency. For instance consider the solution of a problem of heat conduction coupledt with diffusion of a component in a rectangular region, as depicted in figure 21. The boundary condition on side AD is of the form

$$T(x,t) = 100^{\circ}\mathrm{C}\,[4x(L-x)/L^2]\,\sin(5t) \tag{136}$$

where L is the distance AD.

The nodes on side AD are $\{1, 7, 13, 19, 25, 31, 37\}$ and we assume that concentration and temperature are fields 1 and 3 respectively. Here we can take

$$\begin{aligned} a_j &= 4x_j(L-x_j)/L^2], \text{ for } j \text{ in } J \\ \phi(t) &= 100^{\circ}\mathrm{C}\,\sin 5t \end{aligned} \tag{137}$$

Boundary conditions depending on time, as in this example are entered in PETSc-FEM with a `fixa_amplitude` section. The general form is

`fixa_amplitude` *<function>*
<param1> *<val1>*
<param2> *<val2>*

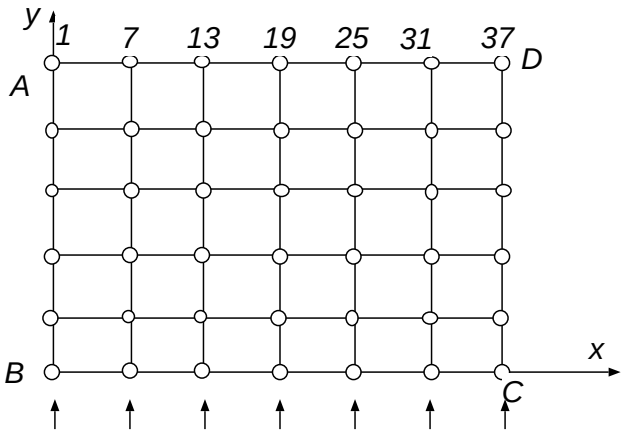


Figure 21: Example with time dependent boundary conditions.

```
...
__END_HASH__
<node1> <dof1> <val1>
<node2> <dof2> <val2>
...
<nodeN> <dofN> <valN>
__END_FIXA__
```

For instance, for the example above it may look like this

```
fixa_amplitude sine
omega 5.
amplitude 1.
__END_HASH__
  0.00000 2
  0.55556 2
  0.88889 2
  1.00000 2
  0.88889 2
  0.55556 2
  0.00000 2
__END_FIXA__
```

14.8.1 Built in temporal functions

The following temporal functions are currently available in PETSc-FEM:

`ramp`: **Double ramp function** See 22.

$$\phi(t) = \begin{cases} \phi_1 & ; \text{ if } t \le t_1 \\ \phi_2 & ; \text{ if } t \ge t_2 \\ \phi_1 + s(t - t_1) & ; \text{ if } t_1 < t < t_2; \end{cases} \tag{138}$$

where the slope s is

$$s = \frac{\phi_2 - \phi_1}{t_2 - t_1} \tag{139}$$

The parameters are

- `start_time`=t_1. (default 0.) The starting time of the ramp.
- `start_value`=ϕ_1. (default 0.) The starting value of the ramp.
- `slope`=s. (default 0.) The slope in the ramp region.
- `end_time`=t_2. (default =`start_time`) The end time of the ramp.
- `end_value`=ϕ_2. (default =`start_value`) The end value of the ramp.

Only one of `slope` and `end_value` must be specified. If the end values are not defined, then they are assumed to be equal to the starting ones. If the end time is equal to the starting time, then the end time is taken as ∞, i.e. a single ramp starting at `start_time`.

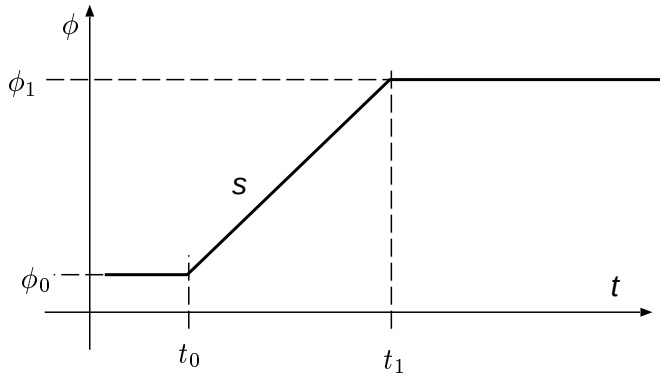


Figure 22: Ramp function

`smooth_ramp`: Smooth double ramp function (hyperbolic tangent) See figure 23.

$$\phi(t) = \frac{\phi_1 + \phi_0}{2} + \frac{\phi_1 - \phi_0}{2} \tanh \frac{t - t_s}{\tau} \tag{140}$$

This function is somewhat analogous to `ramp` in the sense that it goes from a starting constant value to another final one, but smoothly. Beware that the start and end values are reached only for $t \to \pm\infty$.

- `switch_time`=t_s. (default 0.) The time at which ϕ passes by the mean value $\bar{\phi}$ between ϕ_0 and ϕ_1.
- `time_scale`=τ. (default: none) This somewhat represents the duration of the change from the starting value to the end value. During the interval from $t_s - \tau$ to $t_s + \tau$ (i.e. a total duration of 2τ) ϕ goes from $\phi_0 + 0.12\Delta\phi$ to $\phi_0 + 0.88\Delta\phi$.
- `start_value`=ϕ_0. (default 0.) The limit valu for $t \to -\infty$.
- `end_value`=ϕ_1. (default 1.) The limit valu for $t \to +\infty$.

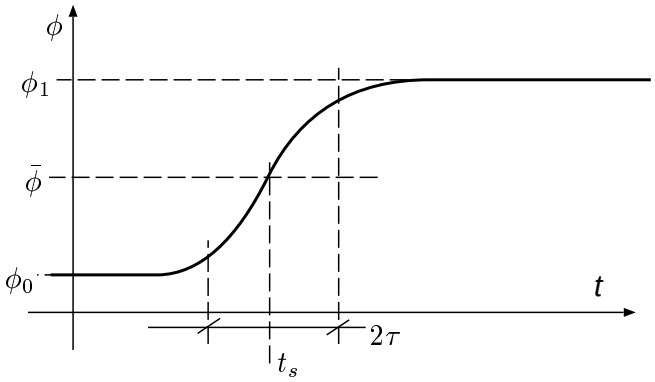


Figure 23: Smooth ramp with the hyperbolic tangent function

`sin`: Trigonometric sine function

$$\phi(t) = \phi_0 + A\,\sin(\omega t + \varphi) \tag{141}$$

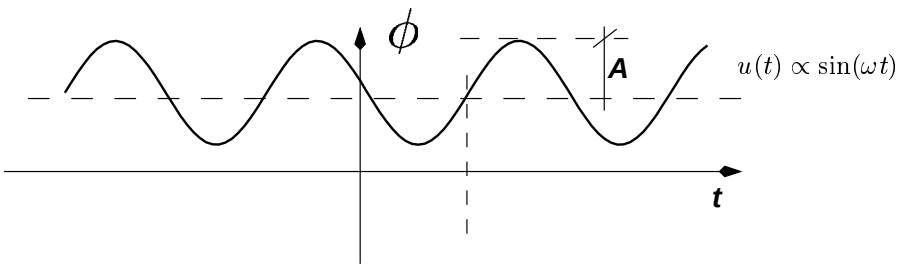


Figure 24: Sine function.

The parameters are

- `mean_val`=ϕ_0. (default 0.)
- `amplitude`=A. (default 1.)
- `omega`=ω. (default: none)
- `frequency`=$\omega/2\pi$. (default: none)
- `period`=$T = 2\pi/\omega$. (default: none)
- `phase`=φ. (default 0.)

Only one of `omega`, `frequency` or `period` must be specified.

`cos`: Trigonometric cosine function

$$\phi(t) = \phi_0 + A\,\cos(\omega t + \varphi) \tag{142}$$

The parameters are the same as for `sin` (see 14.8.1)

`piecewise`**: Piecewise linear function** This defines a piecewise linear interpolation for a given series of time instants t_j (ordered such that $t_j < t_{j+1}$) and amplitudes ϕ_j $(1 < j < n)$ entered by the user. The interpolation is

$$\phi(t) = \begin{cases} 0; & \text{if } t < t_0 \text{ or } t > t_n \\ \phi_k + \frac{\phi_{k+1}-\phi_k}{t_{k+1}-t_k}\, t - t_k; & \text{if } t_k < t < t_{k+1} \end{cases} \tag{143}$$

If `final_time` is defined, then th function is extended periodically with period $t_n - t_1$. The parameters are

- `npoints` (integer) The number n of points to be entered.
- `t` (n doubles) The instants t_j
- `f` (n doubles) The function values ϕ_j
- `final_time` (double) The function is extended from t_n to this instant with period $t_n - t_1$.

`spline`**: Spline interpolated function** This is similar to `piecewise` but data is smoothly interpolated using splines.

- `npoints` (integer) The number n of points to be entered.
- `t` (n doubles) The instants t_j
- `f` (n doubles) The function values ϕ_j
- `final_time` (double) The function is extended from t_n to this instant with period $t_n - t_1$.

`spline_periodic`**: Spline interpolated periodic function** Spline interpolation with `piecewise` may give poor results, specially if you try to match smoothly the beginning and end of the period. `spline_periodic` may give better results, at the cost of being restricted to enter the data function in an equally spaced grid $(t_{j+1} - t_j = \Delta t = \text{cnst})$.

- `npoints` (integer) The number n of points to be entered.
- `period` (double) The period T of the function.
- `start_time` (double, default = 0.) The first time instant t_1. The remaining instants are then defined as $t_j = t_1 + [(j-1)/(n-1)]\, T$.
- `f` (n doubles) The function values ϕ_j at times $\{t_j\}$.

Example: The lines

```
fixa_amplitude spline_periodic
period 0.2
npoints 5
start_time 0.333
ampl_vals  1           0        -1         0.     1.
```

Defines a function with period $T = 0.2$ that takes the values

$$\begin{aligned} \phi(0.333 + nT) &= 1 \\ \phi(0.333 + (n + 1/4)T) &= 0 \\ \phi(0.333 + (n + 1/2)T) &= -1 \\ \phi(0.333 + (n + 3/4)T) &= 0 \end{aligned} \tag{144}$$

Actually, the resulting interpolated function is simply

$$\phi(t) = \cos\left(2\pi\,\frac{t - 0.333}{T}\right) \tag{145}$$

Implementation details: If we define the phase θ as

$$\theta = 2\pi\,\frac{t - t_1}{T} \tag{146}$$

then $\phi(\theta)$ is periodic with period 2π. We can decompose it in two even functions $\phi^{\pm}(\theta)$ as

$$\begin{aligned} \phi^{+}(\theta) &= \frac{\phi(\theta) + \phi(-\theta)}{2} \\ \phi^{-}(\theta) &= \frac{\phi(\theta) - \phi(-\theta)}{2\sin(\theta)} \end{aligned} \tag{147}$$

so that

$$\phi(\theta) = \phi^{+}(\theta) + \sin(\theta)\,\phi^{-}(\theta) \tag{148}$$

As $\phi^{\pm}$ are even function, they may be put in terms of $x = (1 - \cos\theta)/2$. So that x_j and $\phi_j^{\pm}$ are computed and by construction, only one half of the values (say $j = 1$ to $j = m = (n-1)/2 + 1$) are relevant. The values ϕ_0^{-} and ϕ_m^{-} have to be computed specially, since $\sin(\phi_j) = 0$ for them. If we take the limit

$$\phi_0^{-} = \lim_{\theta \to 0} \frac{\phi(\theta) - \phi(-\theta)}{2\sin\theta} \tag{149}$$

then, if linear interpolation is assumed in each interval, it can be shown that

$$\frac{\phi(\theta) - \phi(-\theta)}{2} = \frac{\phi_2 - \phi_{n-1}}{\Delta\theta}\,\theta, \quad \text{for } |\theta| < \Delta\theta \tag{150}$$

Then

$$\lim_{\theta \to 0} \frac{\phi(\theta) - \phi(-\theta)}{2\sin\theta} = \frac{\phi_2 - \phi_{n-1}}{2\Delta\theta} \tag{151}$$

We could take then this value for ϕ_0^{-}, however, this introduces some noise. For instance, if $\phi(\theta) = \sin(\theta)$ then $\phi^{+} \equiv= 0$ and $\phi_j^{-} = 1$ for $j \neq 1, m$, and (151) gives

$$\lim_{\theta \to 0} \frac{\phi(\theta) - \phi(-\theta)}{2\sin\theta} = \frac{\sin\Delta\theta}{\Delta\theta} \tag{152}$$

We introduce, then a *"correction factor"* $\frac{\Delta\theta}{\sin\Delta\theta}$ so that we define

$$\phi_0^{-} = \frac{\phi_2 - \phi_{n-1}}{2\sin\Delta\theta} \tag{153}$$

Analogously, we define

$$\phi_m^{-} = \frac{\phi_{m-1} - \phi_{m+1}}{2\sin\Delta\theta} \tag{154}$$

14.8.2 Implementation details

Temporal boundary conditions are stored in an object of class `Amplitude`. This objects contains a key string identifying the function and a text hash table containing the parameters for that function (for instance `omega`→3.5, `amplitude`→2.3, etc...). Recall that each node/field pair may depend on some free degrees of freedom (those in U in (123)) and some others fixed (those in $\bar{U}$). For those fixed, there is an array containing both an amplitude (i.e. a double) and a pointer to an `Amplitude` object. If the condition don't depends on time, then the pointer is null.

14.8.3 How to add a new temporal function

If none of the built-in time dependent functions fit your needs, then you can add you own temporal function. Suppose you want a function of the form

$$\phi(t) = \begin{cases} 0 & ; t < 0; \\ \frac{t}{(1+t/\tau)} & ; t >= 0; \end{cases} \tag{155}$$

Follow these steps

1. Give a name to it ending in `_function`. We will use `my_own_function` in the following.

2. Declare it with a line like

   ```
   AmplitudeFunction my_own_function;
   ```

   (In the core PETSc-FEM this is done in `dofmap.h`.)

3. Write the definition. (You can find typical definitions in file `tempfun.cpp`). You can use the macro

   `SGETOPTDEF(`<*type*>`,`<*name*>`,`<*default_value*>`);`

   for retrieving values from the hash table defined by the user in the data file.

4. Register it in the temporal function table with a call such as the following, preferably after the call to `Amplitude::initialize_function_table` in the `main()`. For instance

   ```
   Amplitude::initialize_function_table();
   Amplitude::add_entry("smooth_ramp",
              &smooth_ramp_function); // <- add  this line
   ```

   (In the core PETSc-FEM this is done inside the `Amplitude::initialize_function_table`.)

5. Use it in your data files as follows

```
amplitude_function my_own
tau 3.0
__END_HASH__
1 2 3.
...
```

14.8.4 Dynamically loaded amplitude functions

Another possibility to add new amplitude functions is through loading functions at runtime. This avoids re-linkage and modification of the sources. (Note: You need to have the module compiled with the appropriate flag (either the compilation flag `-DUSE_DLEF` or, the Makefile variable `USE_DYNAMICALLY_LOADED_EXTENDED_FUNCTIONS = yes`, either in the `Makefile.base` or the `../Makefile.defs` file.)

These amplitude functions are written in C++ in source files, say `fun.cpp`. The compiled files have extension `.efn` (from *"extension function"*) for instance `fun.efn` in this example. The `Makefile.base` file provides the appropriate rules to generate the compiled `.efn` functions from the source. For each amplitude you want to define you have to write three functions

```
extern "C" void <prefix>init_fun(TextHashTable *thash,void *&fun_data);
extern "C" double <prefix>eval_fun(double t,void *fun_data);
extern "C" void <prefix>clear_fun(void *fun_data) ; // this is optional
```

Prefix may be an empty string, or a non-empty string ending in underscore. Use of non-empty prefix is explained below. The macros `INIT_FUN`, `EVAL_FUN` and `CLEAR_FUN` expand to these definitions. The second function `eval_fun(..)` is that one that defines the relation between the time and the corresponding amplitude. The first `init_fun(...)` one can be used to make some initialization, normally it is called only once. The last one `clear_fun(...)` is called after all calls to `eval_fun(...)`. For simple functions you may not need the init and clear functions, for instance if you want a linear ramp from 0 at $t = 0$ to 1 at $t = 1$, then it can be done with a simple file `fun0.cpp` this:

```
// File: fun0.cpp

#include <math.h>
#include <src/ampli.h>

INIT_FUN {}

EVAL_FUN {
  if (t < 0) return 0.;
  else if (t > 1.) return 1.;
  else return t;
}

CLEAR_FUN {}
```

You then have to do a

`$ make fun0.efn` that will create the `fun0.efn` shared object file. Dynamically loaded functions can be used using the `fixa_amplitude dl_generic` clause and then giving the name of the file with the `ext_filename` option. For instance,

```
<... previous lines here ...>
fixa_amplitude dl_generic
ext_filename "./fun0.efn"
__END_HASH__
 1 1 1.
<... more node/dof/val combinations here ...>
__END_FIXA__
```

You can also have dynamically loaded functions that use parameters loaded via the table of options at run time. For this you are passed the `TextHashTable *` object entered by the user to the `init` function. You then can read parameters from these but in order that the *"init"* function do anything useful it has to be able to pass data to the *"eval"* function. Normally you define a *"struct"* or class to hold the data, create it with `new` or `malloc()` put the data in it, and then pass the address via the *"fun_data"* pointer. Later, in subsequent calls to `eval_fun(...)` it is passed the same pointer so that you can recover the information, previous a static cast. Of course, in order to avoid memory leak you have to free the allocated memory somewhere, this is done in the `clear(...)` function. For instance, the following file defines a ramp function where youcan set the four values t_0, f_0, t_1, f_1

```
#include <math.h>

#include <src/fem.h>
#include <src/getprop.h>
#include <src/ampli.h>

struct  MyFunData {
  double f0, f1, t0, t1, slope;
};

INIT_FUN {
  int ierr;
  // Read values from option table
  TGETOPTDEF(thash,double,t0,0.);
  TGETOPTDEF(thash,double,t1,1.);
  TGETOPTDEF(thash,double,f0,0.);
  TGETOPTDEF(thash,double,f1,1.);

  // create struct data and set values
  MyFunData *d = new MyFunData;
  fun_data = d;
  d->f0 = f0;
  d->f1 = f1;
```

```
  d->t0 = t0;
  d->t1 = t1;
  d->slope = (f1-f0)/(t1-t0);
}

EVAL_FUN {
  MyFunData *d = (MyFunData *) fun_data;
  // define ramp function
  if (t < d->t0) return d->f0;
  else if (t > d->t1) return d->f1;
  else return d->f0 + d->slope *(t - d->t0);
}

CLEAR_FUN {
  // clear allocated memory
  MyFunData *d = (MyFunData *) fun_data;
  delete d;
  fun_data=NULL;
}
```

Another possibility, perhaps more simple, would be to use a global `MyFunData` object but if several `fixa_amplitude` entries that use the same function are used then the data created by the first entry would be overwritten by the second entry.

14.8.5 Use of prefixes

Several functions can be written in the same `.cpp` file using a prefix ending in underscore, for instance if you want to define a `gaussian` function then you define functions with

<*prefix*>=`gaussian_`, for instance

```
extern "C" void gaussian_init_fun(TextHashTable *thash,void *&fun_data);
extern "C" double gaussian_eval_fun(double t,void *fun_data);
extern "C" void gaussian_clear_fun(void *fun_data) ; // this is optional
```

The macros `INIT_FUN1(name)`, `EVAL_FUN1(name)` and `CLEAR_FUN1(name)` do it for you, where `name` is the prefix, with the underscore removed, for instance

```
%
<... headers ...>

struct  MyGaussFunData {
   <... your data here ...>
};

INIT_FUN1(gaussian) {
   <... read data...>
   MyGaussFunData *d = new MyGaussFunData;
```

```
   fun_data = d;
   <... set data in  d ...>
}

EVAL_FUN(gaussian) {
  MyGaussFunData *d = (MyGaussFunData *) fun_data;
  <... use data in  d and define amplitude ...>
}

CLEAR_FUN(gaussian) {
  // clear allocated memory
  MyGaussFunData *d = (MyGaussFunData *) fun_data;
  delete d;
  fun_data=NULL;
}
```

Finally, yet another approach is to have a *"wrapper"* class with methods, `init` and `eval`. The macro `DEFINE_EXTENDED_AMPLITUDE_FUNCTION(class_name)` is in charge of creating and destroying the object. For instance the following file `fun3.cpp` defines two functions `linramp` and `tanh_ramp`.

```
// File fun3.cpp
#include <src/ampli.h>

class  linramp {
public:
  double f0, f1, t0, t1, slope;
  void init(TextHashTable *thash);
  double eval(double);
};

void linramp::init(TextHashTable *thash) {
  int ierr;
  TGETOPTDEF_ND(thash,double,t0,0.);
  TGETOPTDEF_ND(thash,double,t1,1.);
  TGETOPTDEF_ND(thash,double,f0,0.);
  TGETOPTDEF_ND(thash,double,f1,1.);
  slope = (f1-f0)/(t1-t0);
}

double  linramp::eval(double t) {
  if (t < t0) return f0;
  else if (t > t1) return f1;
  else return f0 + slope *(t - t0);
}

DEFINE_EXTENDED_AMPLITUDE_FUNCTION(linramp);
```

```
//---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:---<*>---:
class tanh_ramp {
public:
  double A, t0, delta, base;
  void init(TextHashTable *thash);
  double eval(double);
};

void tanh_ramp::init(TextHashTable *thash) {
  int ierr;
  TGETOPTDEF_ND(thash,double,base,0.);
  TGETOPTDEF_ND(thash,double,A,1.);
  TGETOPTDEF_ND(thash,double,t0,0.);
  TGETOPTDEF_ND(thash,double,delta,1.);
  assert(delta>0.);
}

double  tanh_ramp::eval(double t) {
  if (t < t0) return base;
  else return base + A * tanh((t-t0)/delta);
}

DEFINE_EXTENDED_AMPLITUDE_FUNCTION(tanh_ramp);
```

14.8.6 Time like problems.

The mechanism of passing time to boundary conditions and elemsets may be used to pass any other kind of data, since it is passed as a generic pointer `void *time-data`. This may be used to treat other problems where an external parameter exists, for instance

Continuation problems For instance a NS solution at high Reynolds number may be obtained by continuation in Re, and then we can use it as the external (tome-like) parameter. In general, we can perform continuation in a set of parameters, so that the `time_data` variable should be an array of scalars.

Multiple right hand sides Here the time like variable may be an integer: the number of right hand side case.

15 The compute_prof package

15.1 MPI matrices in PETSc

When defining a matrix in PETSc with "`MatCreateMPIAIJ`" you tell PETSc (see the PETSc documentation)

- How many lines of the matrix live in the actual processoor "`m`".

- For each line of this processor how many non null elements it has in the diagonal block "`d_nnz[j]`"
- For each line of this processor how many non null elements it has in the off-diagonal block "`o_nnz[j]`"

In PETSc-FEM this quantities are computed in two steps.

- Call assemble with "`ijob=COMP_MAT_PROF`" or "`ijob=COMP_FDJ_PROF`" and appropriated "`jobinfo`" in order to tell the element routine which matrix are you computing. This returns a Libretto dynamic array "`*da`" containing the sparse structure of the given matrix.
- function "`compute_prof`" takes "`da`" as argument and fills "`d_nnz`", "`o_nnz`".

15.2 Profile determination

Let us describe first how the sparse profile is determined in the sequential (one processor only) case. We have to determine for each row index i all the column indices j that have a non null matrix entry A_{ij}. The amount of storage needed is a mattern oof concern here. It is allmost sure that we will need at least one integer per each non-null entry. A first attempt is to have a dynamic array for each i index and store their all the connected j indices. In typical applications we have $O(10^4)$ to $O(10^5)$ nodes per processor and the number of connected j indices ranges from 10 to several hundred. In order to avoid this large number of small dynamic arrays growing we store all the indices in a big array, behaving as a sinly linked list. Each entry in the array is composed of two integers ("`struct Node`"), one pointing to the next entry for this row, and other with the value of the row. Consider, for instance, a matrix with the following sparse structure:

$$\mathbf{A} = \begin{bmatrix} * & * & * \\ . & * & . \\ * & . & * \end{bmatrix} \tag{156}$$

Matrix coefficients (i, j) are introduced in the following order (1,1), (2,2), (1,2), (3,1), (1,3), (3,3). The dynamic array for this mesh results in

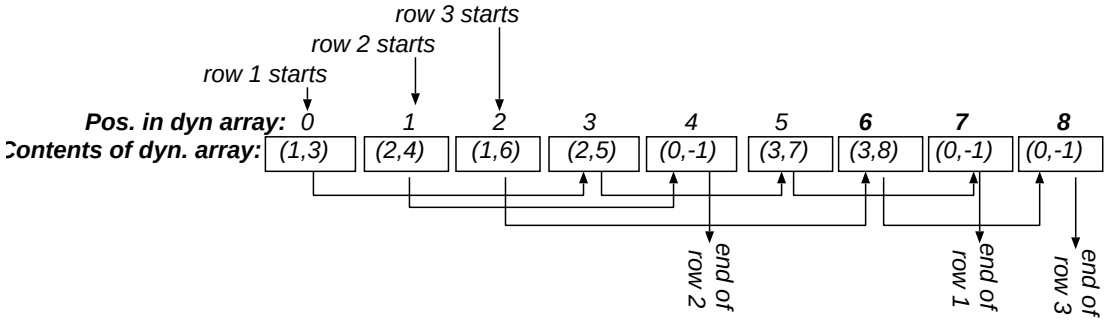

row 3 starts
row 2 starts
row 1 starts
Pos. in dyn array: 0 1 2 3 4 5 6 7 8
Contents of dyn. array: (1,3) (2,4) (1,6) (2,5) (0,-1) (3,7) (3,8) (0,-1) (0,-1)
end of row 2
end of row 1
end of row 3

Figure 25: Structure of darray

For all i, the i-th row starts in position $i-1$. The first component of the pair shows the value of j and the position of the next connected j index. The sequence ends with a terminator $(0,-1)$. In this way, the storage needed is two integers per coefficient with an overhead of two integers per row for the terminator, and there is only one dynamic array growing at the same time.

To insert a new coefficient, say (i,j), we traverse the i-th row, checking whether the j coefficient is already present or not, and if the terminator is found, j is inserted in its place, pointinf to the new terminator, which is placed at the end of the dynamic array.

16 The PFMat class

The `PFMat` class is a matrix class that acts either as a wrapper to the PETSc `Mat` or to other representations of matrix/solvers. Currently there is the basic PETSc class named `PETScMat` and a class called `IISDMat` (for *"Interface Iterative – Sub-domain Direct"*, method) that has a special solver that solves the linear system by solving iteratively over the interface nodes, while solving with a direct method in the sub-domain interiors (this is commonly referred as the *"Domain Decomposition Method"*).

16.1 The PFMat abstract interface

The `create(...)` member should create the matrix from the matrix profile `da` and the `dofmap'`. For the `PETScMat` matrix class it calls the old `compute_prof` routine calculating the `d_nnz` and `o_nnz` arrays, and calling the `MatCreate` routine. For the `IISD` matrix it has to determine which dofs are in the local blocks and create the appropriate numbering.

The `set_value` member is equivalent to the 'MatSetValues' routine and allows to enter values in the matrix. For the `IISDMat` class it sets the value in the appropriate block (A_{LL}, A_{IL}, A_{LI} or A_{II} PETSc matrices). In addition, for the A_{LL} (*"local-local"* block) it has to *"buffer"* those values that are not in this processor (this can happen when dealing with periodic boundary conditions, or bad partitionings, for instance, an element that is connected to all nodes that belong to other processor. This last case is not the most common but it can happen.

Once you have filled the entries in the matrix you have to call the `assembly_begin()` and `assembly_end()` members (as in PETSc).

The `solve(...)` member solves a linear ssytem associated to the given operator.

The `zero_entries()` is also the counterpart of the corresponding PETSc() routine. The `build_sles()` member creates internally the SLES needed by the solution (included the preconditioner). It takes as an argument a `TextHashTable` from where it takes a series of options. The `destroy_sles()` member has to be called afterwards, in order to destroy it (and free space). The `monitor()` member allows the user to redefine the convergence monitoring routine.

The `view()` member prints operator information to the output.

16.2 IISD solver

Let's consider a mesh as in figure 26, partitioned such that a certain number of elements and nodes belong to processor 0 and others to processor 1. We assume that one unknown

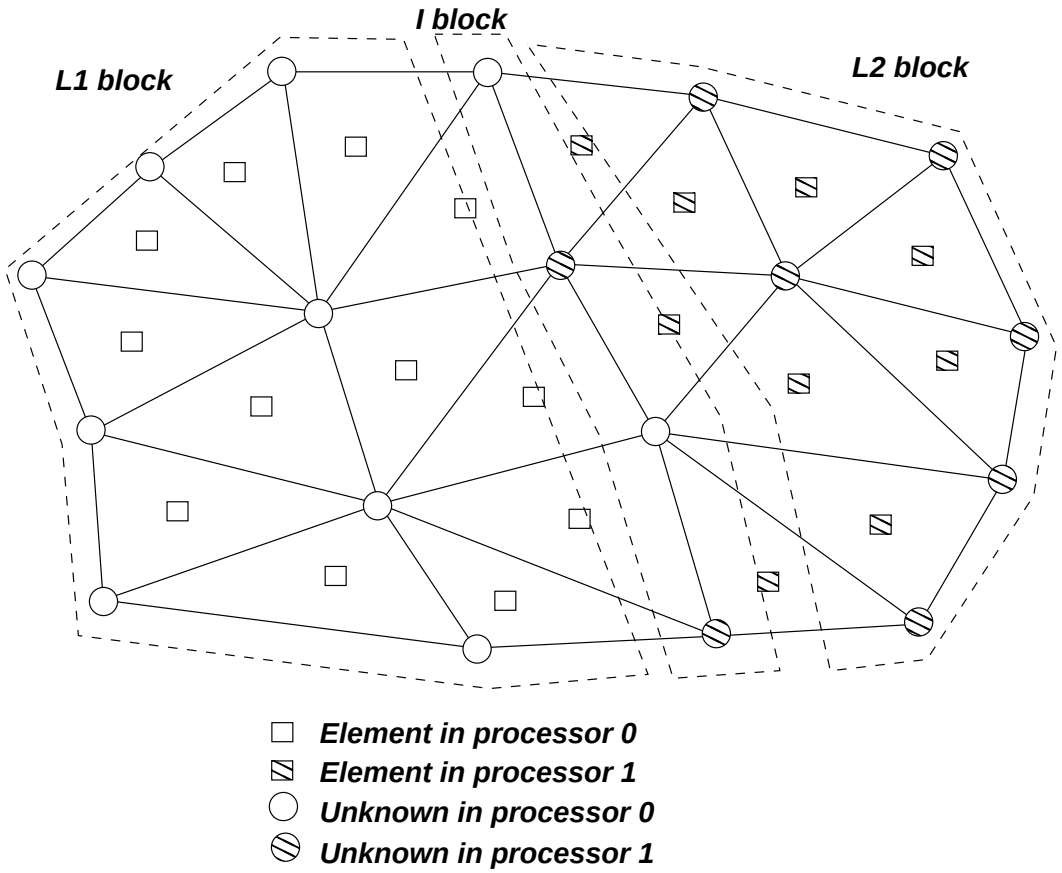


Figure 26: IISD deccomposition by subdomains

is associated to each node and no Dirichlet boundary conditions are imposed so that each node corresponds to one unknown. We split the nodes unknowns in three disjoint subsets $L_{1,2}$ and I such that the nodes in L_1 are not connected to those in L_2 (i.e. they not share an element, and then, the FEM matrix elements $A_{i,j}$ and $A_{j,i}$ with $i \in L_1$ and $j \in L_2$ are null. The matrix is split in blocks as follows

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} \mathbf{A}_{LL} & \mathbf{A}_{LI} \\ \mathbf{A}_{IL} & \mathbf{A}_{II} \end{bmatrix} \\ \mathbf{A}_{LL} &= \begin{bmatrix} \mathbf{A}_{00} & 0 \\ 0 & \mathbf{A}_{11} \end{bmatrix} \\ \mathbf{A}_{LI} &= \begin{bmatrix} \mathbf{A}_{0I} & \mathbf{A}_{1I} \end{bmatrix} \\ \mathbf{A}_{IL} &= \begin{bmatrix} \mathbf{A}_{I0} \\ \mathbf{A}_{I1} \end{bmatrix} \end{aligned} \tag{157}$$

Now consider the system of equations

$$\mathbf{A}\mathbf{x} = \mathbf{b} \tag{158}$$

which is split as

$$\begin{aligned} \mathbf{A}_{LL}\,\mathbf{x}_L + \mathbf{A}_{LI}\,\mathbf{x}_I &= \mathbf{b}_L \\ \mathbf{A}_{IL}\,\mathbf{x}_L + \mathbf{A}_{II}\,\mathbf{x}_I &= \mathbf{b}_I \end{aligned} \tag{159}$$

Now consider eliminating $\mathbf{x}_L$ from the first equation and replacing in the second so that, we have an equation for $\mathbf{x}_I$

$$\begin{aligned} (\mathbf{A}_{II} - \mathbf{A}_{IL}\,\mathbf{A}_{LL}^{-1}\,\mathbf{A}_{LI})\,\mathbf{x}_I &= (\mathbf{b}_I - \mathbf{A}_{IL}\,\mathbf{A}_{LL}^{-1}\,\mathbf{b}_L) \\ \tilde{\mathbf{A}}\,\mathbf{x}_I &= \tilde{\mathbf{b}}_I \end{aligned} \tag{160}$$

We consider solving this system of equations by an iterative method such as GMRES, for instance. For such an iterative method, we have only to specify how to compute the modified right hand side $\tilde{\mathbf{b}}$ and also how to compute the matrix-vector product $\mathbf{y} = \tilde{\mathbf{A}}\,\mathbf{x}$. Computing the matrix product involves the following steps

1. Compute $\mathbf{y} = \mathbf{A}_{II}\,\mathbf{x}$
2. Compute $\mathbf{w} = \mathbf{A}_{LI}\,\mathbf{x}$
3. Solve $\mathbf{A}_{LL}\,\mathbf{z} = \mathbf{w}$ for $\mathbf{z}$
4. Compute $\mathbf{v} = \mathbf{A}_{IL}\,\mathbf{z}$
5. Add $\mathbf{y} \leftarrow \mathbf{y} - \mathbf{v}$

involving three matrix products with matrices $\mathbf{A}_{II}$, $\mathbf{A}_{IL}$ and $\mathbf{A}_{LI}$ and to solve the system with $\mathbf{A}_{LL}$. As the matrix $\mathbf{A}_{LL}$ has no elements connecting unknowns in different processors the solution system may be computed very efficiently in parallel.

16.2.1 Interface preconditioning

To improve convergence of the interface problem(160) some preconditioning can be introduced. As the interface matrix is never built, true Jacobi preconditioning (i.e. using the diagonal part of the Schur complement, $\mathbf{P} = \operatorname{diag}\tilde{\mathbf{A}}$, where $\mathbf{P}$ is the preconditioning matrix), can not be used. However we can use the diagonal part of the interface matrix ($\mathbf{P} = \operatorname{diag}\mathbf{A}_{II}$) matrix or even the whole interface matrix ($\mathbf{P} = \mathbf{A}_{II}$). Using the diagonal preconditining helps to reduce bad conditioning due to refinement and inter-equation bad scaling. If the whole interface matrix is used ($\mathbf{P} = \mathbf{A}_{II}$) then the linear system $\mathbf{A}_{II}\mathbf{w} = \mathbf{x}$ has to be solved at each iteration. This can be done with a direct solver or iteratively. In the 2D case the connectivity of the interface matrix is 1D or 1D like, so that the direct option is possible. However in 3D the connectiviy is 2D like, and the direct solver is much more expensive. In addition the interface matrix is scattered among all processors. The iterative solution is much more appealing since the fact that the matrix is scattered among processors is not a problem. In addition the interface matrix is usually diagonally dominant, even when the whole matrix is not. For instance for the Laplace equation in 2D the stencil of the interface matrix on a planar interface in a homogeneous grid of step h with bilinear quad elements is $[-18-1]/3h^2$. Such matrix has a condition number which is independent of h and is asymptotically $\kappa(A_{II}) = 5/3$. In the case of using triangular elements (by the way, this is equivalent to the finite difference case) the stencil

is $[-14-1]/2h^2$ whose condition number is $\kappa(A_{II}) = 3$. This low condition number also favors the use of an iterative method. However a disdvantage for the iterative solution is that non-stationary iterative solvers (i.e., those where the next iteration x^{k+1} doesn't depend only on the previous one: $x^{k+1} = f(x^k)$, like CG or GMRES) can not be nested inside other non-stationary method, unless the inner preconditioning loop is iterated to a very low error. This is because the conjugate gradient directions will lose orthogonality. But using a very low error bound for the preconditioning problem may be expensive so that non-stationary iterative methods are discarded. Then, the use of Richardson iteration is suggested. Sometimes this can diverge and some under-relaxation must be used. Options controlling iteration for the preconditioning problem can be found in section § 4.4.3.

16.3 Implementation details of the IISD solver

- Currently unknowns and elements are partitioned in the same way as for the PETSc solver. The best partitioning criteria could be different for this solver than for the PETSc iterative solver.

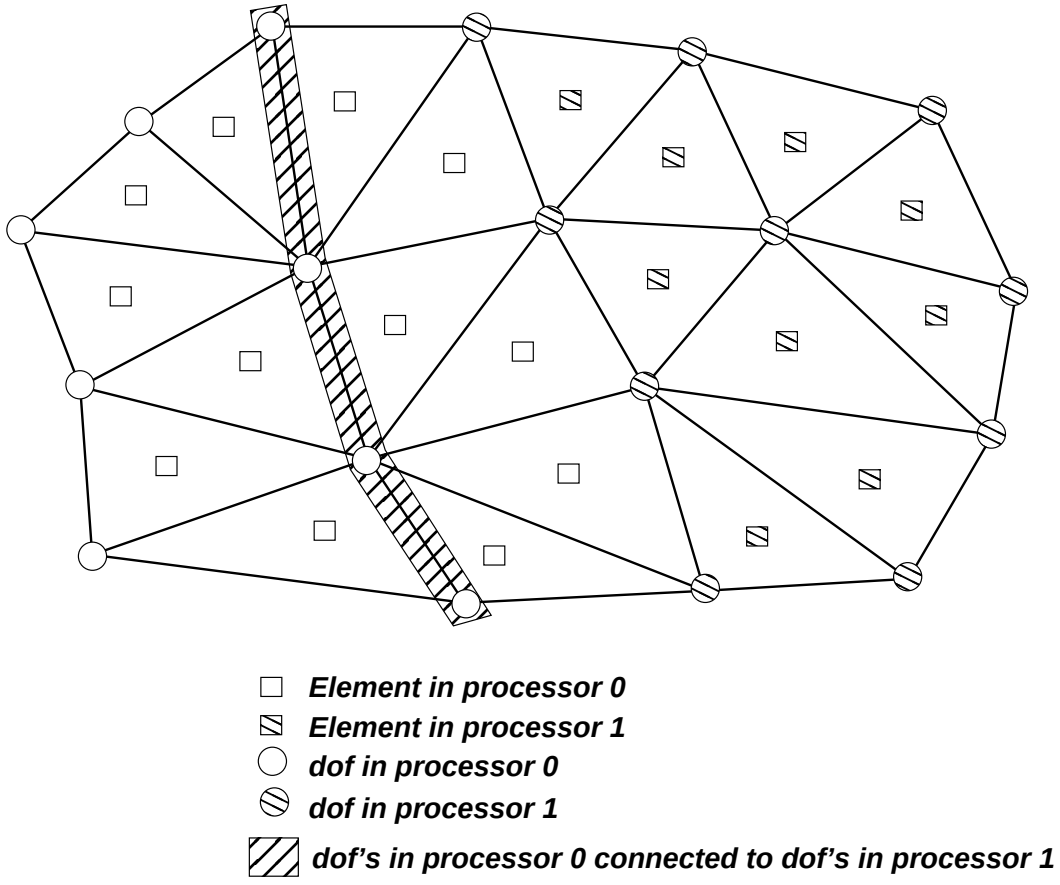

Element in processor 0
Element in processor 1
dof in processor 0
dof in processor 1
dof's in processor 0 connected to dof's in processor 1

Figure 27: IISD deccomposition by subdomains. Actual decomposition.

- Selecting *"interface"* and *"local"* dof's: One strategy could be to mark all dof's that are connected to a dof in other processor as *"interface"*. However this could lead to

an *"interface"* dof set twice larger in the average than the minimum needed. As the number of nodes in the *"interface"* set determines the size of the interface problem (160) it is clear that we should try to choose an interface set as small as possible.

In `read_mesh()` partitioning is done on the dual graph, i.e. on the elements. Nodes are then partitioned in the following way: A node that is connected to elements in different processors is assigned to the highest numbered processor. Referring to the mesh in figure 26 and with the same element partitioning all nodes in the interface would belong to processor 1, as shown in figure 27.

Now, if a dof i is connected to a dof j on other processor we mark as *"interface"* that dof that belongs to the highest numbered processor. So, in the mesh of figure 27 all dof's in the interface between element sub-domains are marked to belong to processor 1. The nodes in the shadowed strip belong to processor 0 and are connected to nodes in processor 1 but they are not marked as *"interface"* since they belong to the lowest numbered processor. Note that this strategy leads to an interface set of 4 nodes, whereas the simpler strategy mentioned first would lead to an interface set of 4 (i.e. including the nodes in the shadowed strip), which is two times larger.

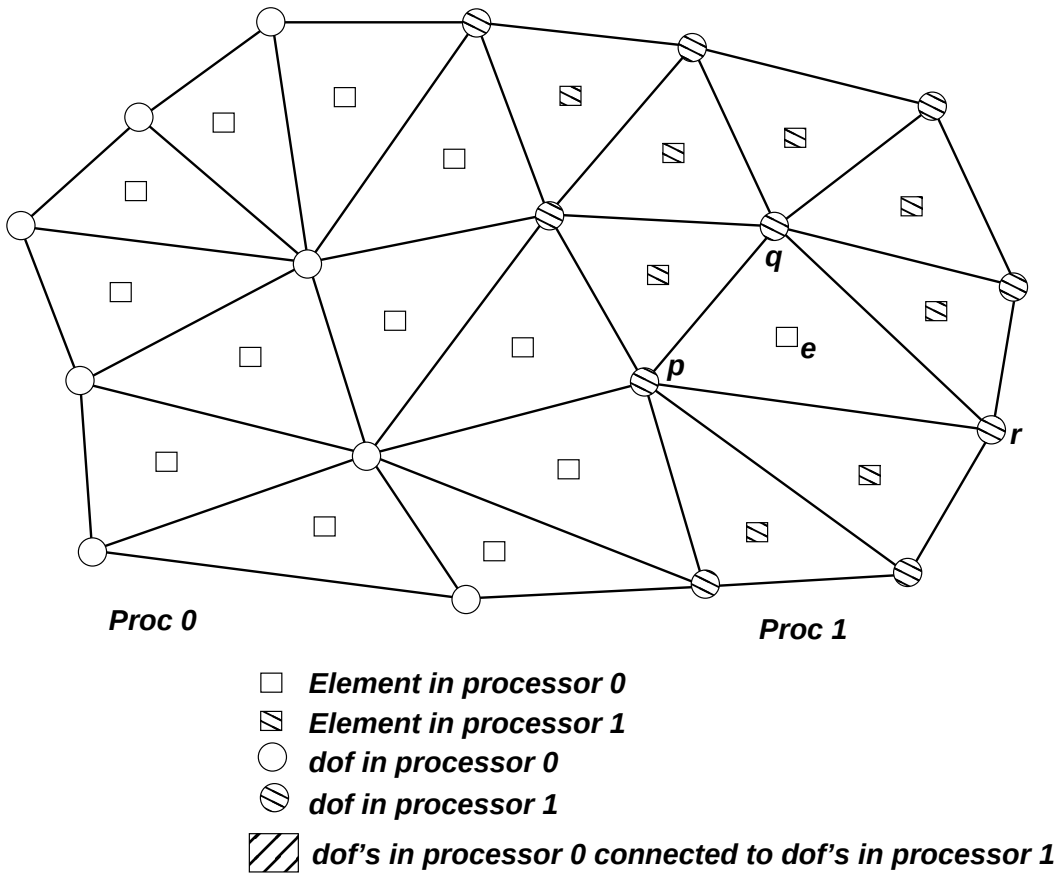


Figure 28: Non local element contribution due to bad partitioning

- The `IISDMat` matrix object contains three `MPI` PETSc matrices for the A_{LI}, A_{IL} and A_{II} blocks and a sequential PETSc matrix in each processor for the local part

of the A_{LL} block. The A_{LL} block must be defined as sequential because otherwise we couldn't factorize it with the LU solver of PETSc. However this constrains that `MatSetValues` has to be called in each processor for the matrix that belongs to its block, i.e. elements in a given processor shouldn't contribute to A_{LL} elements in other processors. Normally, this is so, but for some reasons this condition may be violated. One, is periodic boundary conditions and constraints in general (they are not taken into account for the partitioning). Another reasons is very bad partitioning that may arise in some not so common situations. Consider for instance figure 28. Due to bad partitioning a rather isolated element e belongs to processor 0, while being surrounded by elements in processor 1. Now, as nodes are assigned to the highest numbered processor of the elements connected to the node, nodes p, q and r are assigned to processor 1. But then, nodes q and r will belong to the local subset of processor 1 but will receive contributions from element e in processor 0. However, the solution is not to define this matrices as PETSc because, so far, PETSc doesn't support for a distributed LU factorization. The solution we devised is to store those A_{LL} contributions that belong to other processors in a temporary buffer and after, to send those contributions to the correct processors directly with MPI messages. This is performed with the DistMatrix object `A_LL_other`.

16.4 Efficiency considerations

The uploading time of elements in PETSc matrices can be significantly reduced by using *"block uploading"*, i.e. uploading an array of values corresponding to a rectangular submatrix (not necessarily being contiguous indices) instead of uploading each element at a time. The following code snippets show the both types of uploading.

```
// ELEMENT BY ELEMENT UPLOADING
// The global matrix
Mat A;
// row and column indices (both of length 'nen')
int *row_indx,*col_indx;
// Elementary matrix (size 'nen*nen')
double *Ae;
// ... define row_indx, col_indx and fill Ae
for (int j=0; j<nen; j++) {
  for (int k=0; k<nen; k++) {
    ierr = MatSetValue(A,row_int[j],col_indx[k],Ae[j*nen+k],ADD_VALUES);
  }
}

// BLOCK UPLOADING
// ... same stuff as before ...
ierr = MatSetValues(A,nen,row_int[j],nen,col_indx[k],Ae,ADD_VALUES);
```

In PETSc-FEM, the computed elemental matrices can be uploaded in the global matrices with both methods, as selected with the `block_uploading` global option (set to 1 by default, i.e. use block uploading). However, in some cases block uploading can be actually

slower due to the use of *"masks"*. A mask is a matrix of the same size as the elemental matrix with 0's or 1's indicating whether some coefficients are (structurally, i.e. not for a particular state) null. Moreover, the mask do not depends on the particular element, but it is rather a property of the terms being evaluated in the Jacobian.

For instance, for the Navier-Stokes equations the Galerkin term only has non null coefficients for the velocity unknowns in the continuity equation, while the pressure gradient term only has coefficients for the pressure unknowns in the momentum equations. Both terms together have a mask as shown in figure 29. When the application writer codes such a term, he defines the mask. At the moment of uploading the elements, if `block_uploading` is in effect, then PETSc-FEM computes the *"envelope"* of the mask, i.e. the rectangular mask that contains the mask in order to make just one call to `MatSetValues`. In this case, the envelope is just a matrix filled with 1's, so that block uploading pays the benefit of using the faster `MatSetValues` routine, with the cost of loading much more coefficients than the original mask. In addition, the PETSc matrix will be bigger, with the corresponding increase in RAM demand and CPU time in computing factorizations (IISD solver) and matrix/vector products (PETSc solver). (In a future, such a combination of terms will be loaded more efficiently with two calls to `MatSetValues`.)

The conclusion is that, if the terms to be loaded have a very sparse structure but a dense envelope, then may be block uploading is slower. (The worst case is a diagonal-like mask.) Note that also, it's not sufficient to have a sparse structure of the elemental matrix, but also the application writer has to compute and return the mask. Finally, note that you can always check whether block uploading is faster or slower by activating the time statistics for the elemset (the `report_consumed_time`) and run a large example with both kinds of uploading.

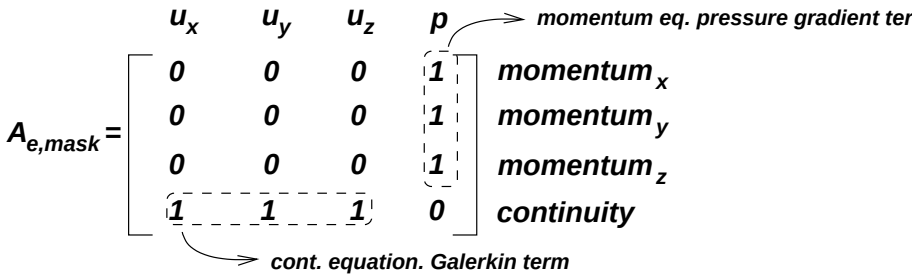

u_x u_y u_z p
momentum eq. pressure gradient ter
$A_{e,mask}$ =
0 0 0 1 momentum$_x$
0 0 0 1 momentum$_y$
0 0 0 1 momentum$_z$
1 1 1 0 continuity
cont. equation. Galerkin term

Figure 29: Example of mask for the continuity equation (Galerkin term) in the Navier-Stokes equations

17 The DistMap class

This class is an STL `map<Key,Val>` template container, where each process can insert values and, at a certain point a `scatter()` call sends contributions to the corresponding processor.

17.1 Abstract interface

To insert or access values one uses the standard `insert()` member or the `[]` operator of the basic class. The `processor(iterator k)` member (to be defined by the user of the template) should return the process rank to which the pair `(key,val)` pointed by `k` should belong. After calling the `scatter()` member by all processes, all entries are sent to their processors. In order to send the data across processors, the user has to define the `size_of_pack()` and `pack()` routines. The `pack()` and `unpack()` functions in the `BufferPack` namespace can help to do that for basic types (i.e. those for which the `sizeof()` operator and the `libc memcpy()` routine work). Finally the `combine` member defines how new entries that have been sent from other processors have to be merged in the local object.

17.2 Implementation details

When calling the `scatter` member, each entry in the basic container is scanned and if it doesn't belong to the processor it is buffered (using the `pack` member) to a buffer to be sent to the other processor. Once all the data to be sent is buffered, the entries are scanned again and the entries that have been buffered are deleted.

The buffers are sent to the other processors following a strategy preventing deadlock in $n_{\text{proc}} - 1$ stages, where n_{proc} is the number of processors. In the k-th stage, data is sent from processor j to processor $(j + k)\%n_{\text{proc}}$ where % is the remainder operator as in the C language. In each stage, all processors other than the server do first a `MPI_Recv()` and after a `MPI_Send()`, while the server does in the other sense, i.e. first a `MPI_Send()` and after a `MPI_Recv()`, initiating the process

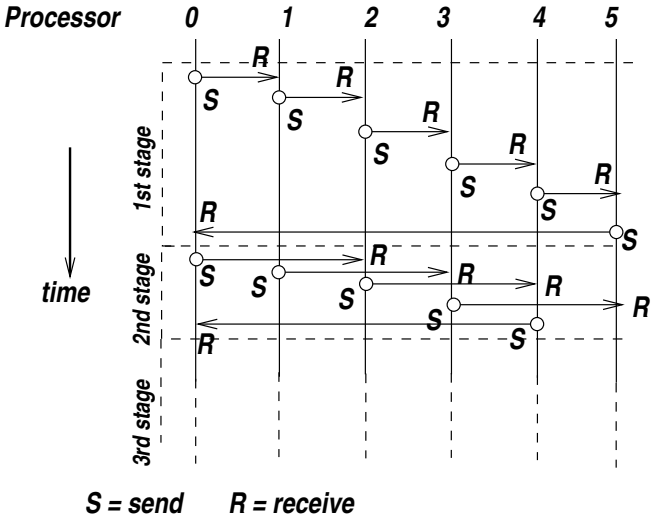


Figure 30: Simple scheduling algorithm for transferring data among processors

This is a rather inefficient strategy because at each stage all sending is tied to the previous one, making the whole process $\sim {n_{\text{proc}}}^2 T$) where T is the time needed to send a typical individual buffer from one process to other.

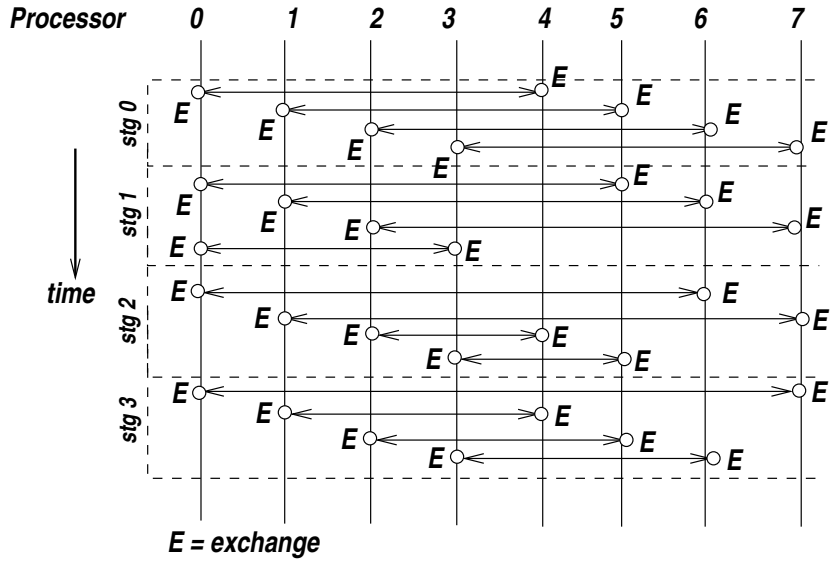


Figure 31: Improved scheduling algorithm for transferring data among processors

An improved strategy consists in a multilevel scheduling algorithm. Assume first that n_{proc} is even and divide processors in subsets $S_1 = \{0, 1, \ldots, n_{\text{proc}}/2 - 1\}$, $S_2 = \{n_{\text{proc}}/2, n_{\text{proc}}/2 + 1, \ldots, n_{\text{proc}} - 1\}$. We first exchange all information between processors in S_1 with those in S_2 in $n_{\text{proc}}/2$ stages. In stage 0 (see figure 31) processor 0 exchanges data with $n/2$, 1 with $n_{\text{proc}}/2 + 1$, ... and $n/2 - 1$ with n_{proc}. i.e. processor 0 calls `exchange`$(n/2)$ and processor $n/2$ calls `exchange`(0), the code of procedure `exchange()` is shown in the procedure below. In the stage 1 processor 0 exchanges with $n_{\text{proc}}/2 + 2$, 1 with $n_{\text{proc}}/2 + 3$, $n_{\text{proc}}/2 - 2$ with n_{proc} and $n_{\text{proc}}/2 - 1$ with $n_{\text{proc}}/2$, i.e. processor j exchanges with $(n/2) + (j + 1)\%(n_{\text{proc}}/2)$. In general, in stage k processor j exchanges with processor $(n/2) + (j + k)\%(n_{\text{proc}}/2)$. Note that in each stage all communications are paired and can be performed simultaneously, so that each stage can be performed in $2T$ (sending plus receiving). This $n_{\text{proc}}/2$ stages take then a total of $2(n_{\text{proc}}/2)T = n_{\text{proc}}T$ secs. Now, all communication between processors in S_1 and S_2 has been performed, it only remains to perform the communication between processors in S_1 and processors in S_2. But then, we can apply this idea recursively and divide the processors in S_1 in two subsets S_{11} and S_{12} with $n_{\text{proc}}/4$ each (let's assume that the number of processors is a power of 2, $n_{\text{proc}} = 2^m$), with a required time of $(n_{\text{proc}}/2)T$. Applying the idea recursively we arrive to an estimation of a total time of

$$\begin{aligned} T(n) &= [n_{\text{proc}} + (n_{\text{proc}}/2) + \ldots + 1]T \\ &= (2n_{\text{proc}} - 1)T \\ &\sim O(2n_{\text{proc}}T) \end{aligned} \tag{161}$$

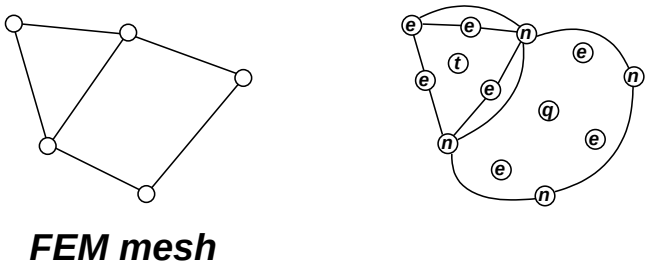
Then, it is significantly better than the previous algorithm.

```
// Scheduling algorithm for exchanging data between processors
void exchange(j) {
  if (myrank>j) {
```

```
    // send data to processor j ...
    // receive data from processor j ...
  } else {
    // send data to processor j ...
    // receive data from processor j ...
  }
}
```

17.3 Mesh refinement

[Warning: This is a work in progress.]



FEM mesh

Figure 32: FEM mesh and graph representation

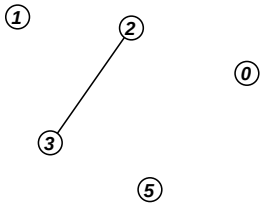


Figure 33: Geomtrical objets: 5 nodes and an edge

We conceive a mesh as a graph, connecting nodes and other higher dimension entities based on these nodes. Consider for example five nodes (labeled from 0 to 40 and an edge connecting two of these nodes as in figure 33. In total, there are 6 *"geometrical objects"* (5 nodes and the edge) in the figure. In order to identify higher order objects like the edge, we could add a new index for it, say index 5, but instead we can associate the edge with the connected pair of node indices, `(edge 2 3)`. Note that, if we consider that the edge has no orientation, then the sequence must be considered as a *"set"*, i.e. `(edge 2 3)` = `(edge 3 2)`, (Lisp-like *"S-exp"* expressions will be used throughout in order to describe objects) whereas if oritentation matters, then `(edge 2 3)` $\neq$ `(edge 3 2)`. We could, then define geometrical objects of type `edge` as unordered sequences of two node indices, while the type `ordered-edge` is associated with ordered sequences of two nodes. We say that the set of permutations that leave invariant `edge` objects is `((0 1) (1 0))`, whereas for the oriented edge is only the identity permutation `((0 1))`. For larger objects, the set of permutations that leave invariant the node sequence that defines the object is more complex than that, it doesn't reduce to the special case of ordered and unordered

sequences as for edges. Consider for instance the case of triangles. For an oriented triangle, the set of nodes that leave invariant the triangle is `((0 1 2) (1 2 0) (2 0 1))`, i.e. shifts clockwise and counter-clockwise of the node sequence, whereas for unordered objects the sequences are the same `((0 1 2) (1 2 0) (2 0 1) (0 2 1) (1 0 2) (2 1 0))`. In the last case, the permutations that leave invariant the object are the whole set of 6 permutations for 3 objects, so that in this case we can say that unordered triangles can be represented as unordered sets of three nodes. But for the unordered triangle edge `(a b c)` is the same that `(b c a)`, so that associating unordered triangles with unordered sequences does not take into account the rotational symmetry.

17.3.1 Symmetry group generator

The set of permutations `(perm SHAPE)` that leave the geometrical object `SHAPE` invariant are a *"group"*, that means that if two permutations p, q belong to `(perm SHAPE)`, then the composition `pq`, i.e. the permutation resulting of applying first q and then p, also belong to the group, as well as the product `qp`. In general, this is a finite group.

As a consequence, if we have two elements p and q of a group G, then pq, qp, p^2q, pqp, pq^2... all belong to G. Given a set of permutations $S \subset G$, the set pS formed by the *left* product of all the elements of S with p belongs to G. The same applies to Sp, qS and Sq. So that, starting with the set $S_0 = \{1\}$, where 1 is the identity permutation, we can generate recursively a larger set of symmetries by forming $S_1 = \{S_0, pS_0, S_0p, qS_0, S_0q\}$. As the total number of permutations in the group is finite (at most $n!$, where n is the number of nodes in the object), applying recursively the relation above will end with a set of permutations H that is itself a group included in G. We will call it the group *"generated"* by p an q. This can be applied to any number of generators p, q, r, s, ...

For instance, the symmetries for the oriented triangle can be generated with the permutation $p =$`(1 2 0)`, since `(2 0 1)` can be generated as p^2. The symmetries for the unoriented triangle can be derived from generators `(1 2 0)` (rotation) and `(0 2 1)` (inversion). The most relevant geometrical objects and their symmetries are

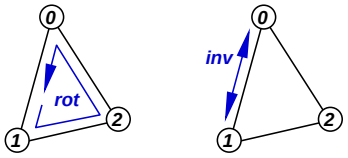


Figure 34: Generators for triangle

- Unoriented triangles are described by numbering their nodes in any way. Their symmetries are all the permutations ($6 = 2!$ en total) and are generated by a rotation and an inversion.
- Oriented triangles are described by numbering their nodes in a specified direction. Their symmetries are 3 en total generated by a rotation.
- Unoriented quadrangles are described by numbering their nodes in clockwise or counterclockwise rotation. Symmetry generators: are `(1 2 3 0)` (rotation) and `(0 3 2 1)` (inversion), there is a total of 8 permutations.

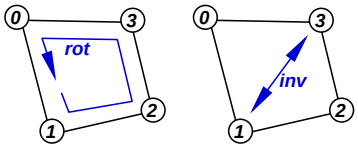


Figure 35: Generators for quadrangle

- Oriented quadrangles are identical to unoriented quadrangles but do not include the inversion (4 rotations).

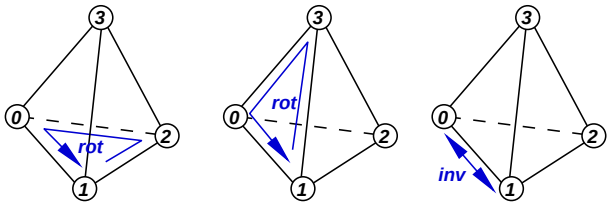


Figure 36: Generators for tetras

- Unoriented tetrahedra are described by numbering one of the faces, and then the oposite node. Symmetries are generated by permutation of any two of the faces, for instance `(1 2 0 3)` and `(3 0 2 1)` and inversion `(1 0 2 3)` (24 permutations in total).
- Oriented tetrahedra is identical to unoriented tetrahedra without inversion (24 permutations in total).

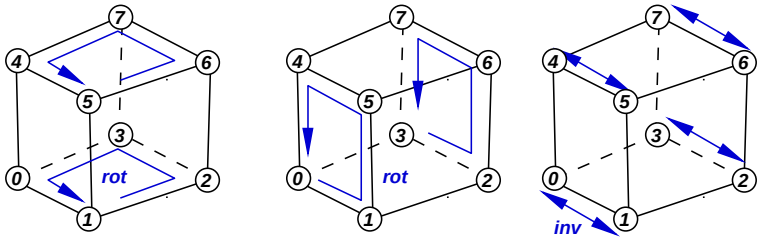


Figure 37: Generators for hexas

- Unoriented hexas are described by numbering one of the faces as a quad, and then the opposite face in correspondence with the first face. Symmetries are generated by 90° rotations for any two of the faces (not opposite), for instance `(1 2 3 0 5 6 7 0)` and `(1 5 6 2 0 4 7 3)` and inversion `(1 0 2 3)` (232 permutations in total).
- Oriented hexahedra is identical to unoriented hexahedra without inversion (112 permutations in total).
- Unoriented prisms are described by numbering one of the triangular faces as a triangle, and then the opposite face in correspondence with the first face. Symmetries are generated by 180° rotations of any two of the quad faces (`(4 3 5 1 0 2)` for instance), a 120° rotation of any of the triangular face (`(1 2 0 4 5 3)` for instance) and an inversion (`(1 0 2 4 3 5)` for instance). There are 12 permutations in total.

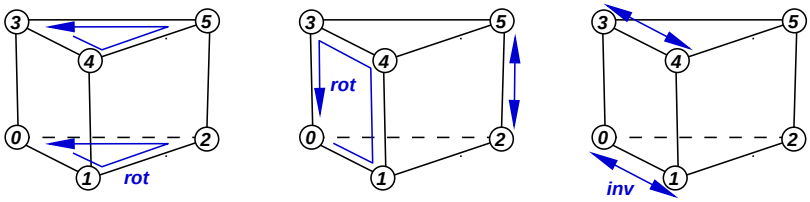


Figure 38: Generators for prisms

- Oriented hexahedra is identical to unoriented hexahedra without inversion (6 permutations in total).

17.3.2 Canonical ordering

One of the most common operations when manipulating these geometrical objects is, given two geometrical objects of the same type, to determine whether they represent the same object. The brute force solution is to apply all the permutations to one of them and check if one of the permuted indices coincide with the other node sequence.

17.4 Permutation tree

In order to make it more efficient we can store all the permutations for a given shape in a tree like fashion. Consider, for instance, the oriented tetrahedral shape. Their permutations are the following:

```
(0 1 2 3)
(0 2 3 1)
(0 3 1 2)
(1 0 3 2)
(1 2 0 3)
(1 3 2 0)
(2 0 1 3)
(2 1 3 0)
(2 3 0 1)
(3 0 2 1)
(3 1 0 2)
(3 2 1 0)
```

We can describe the generation of a new node numbering in the following way. First, we can take any of the nodes as the first node of the new numbering. These is seen from the fact that all the indices (0 to 3) are present in the first column of the table above. If we chose node 2 as the first index, then we can chose any of the reamining as the second node (0 1 3). Once we choose the second index (say for instance 1) there is only one possibility for the reamining two: the numbering `(3 1 0 2)`. The remaining possibility `(3 1 2 0)` would gnerate an inverted triangle. Part of the tree, is shown in 39. Every possible permutation is a *"path"* in the tree from the first node at level 0, to the last node, which is a leave.

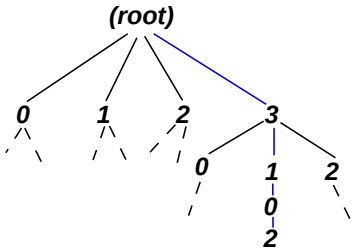


Figure 39: Tree representing all the permutations for the ordered tetra geometry.

Now, given two possible node orderings for a given geometrical object, and the tree that describes the permutations for its shape, we simply have to follow the path that makes one of them to fit in the other. If we arrive to an internal node without possibility to follow, then the geometrical objects are distinct. If we reach a leave, then the objects are the same.

17.5 Canonical ordering

Another possibility to determine whether two node oderings are congruent is to have a uniquely determined ordering that can be computed from the node sequence itself. We we call this ordering the *"canonical"* ordering for the geometrical object. If this is possible, then given two orderings we can bring both of them to the canonical ordering and then compare them as plain sequences. One possibility is to take the caonical order as that one that gives the lower node sequence, in lexicographical order.

This has the advantage that one the canonical orderign is known for both objects, the comparison is very cheap. Also, it can be used as a comparison operator for sorting geometrical objects, i.e. the order between two objects is the lexicographical order between the two node sequences in its canonical form.

The canonical order can be computed efficientlyusing the permutation tree described above.

17.6 Object hashing

Another useful technique for comparing objects is by comparing first some scalar function of the node indices. For instance we can compute a *"hash-sum"* for the object `go` as

$$H(\texttt{go}) = \sum_{j=0}^{n-1} h((\texttt{nodegoj})) \tag{162}$$

where `(node go j)` is the `j` node of object `go`. H is the hash-sum for the geometrical object `go`, whereas h is a *"scalar hashing function"*. A very simple possibility is to take h as the identity $h(x) = x$. In that case the hash of the object is simply the sum of their node indices. The idea is that, even for this simple hashing function we can compare first the hash-sums of two objects for determining if they are equal. If there is a high probability of the objects being unequal, then in most cases this simple check will suffice. If the hash-sum are equal, then the full comparision procedure, as described above, must be performed. Note that the hash-sum is (and must be) independent of the ordering.

In order to reduce the probability that unequal objects give the same hash-sum we can device better hashing functions. Two kind of hashing functions will be discussed: functions that hashes a sequence of numbers (*"hash-sequence functions"*) taking in account the ordering of the sequences and functions that hash the sequences in a way infependent of the ordering (*"hash-sum functions"*).

Consider first the hash-sequence functions. If we consider sequences of integers, then the hash functions should return a scalar value for each sequence of integers, in such a way that if two sequences are distinct (i.e. they have distinct numbers, or the same numbers in differente order) they should have different hashing functions. We restrict us to 32-bit integers, unsigned integers are bounded by 2^{32} and there are 2^{32n} distinct sequences of n integers, so that it is impossible to have such un hashing function. If for two distinct sequences we have the same hash values, then we said thar there is a *"collision"* in the hashing process. We, then, try to have a hashing functions that minimizes the number of collisions. Suppose we consider the set of number sequences 32-bit integers of length n. We have $N = 2^{32n}$ sequences and $M = 2^{32}$ hash values, and in the best case we would have $N/M = 2^{32(n-1)}$ sequences for each hash value. If we generate m distinct (random) sequences and $m \ll N$ then, if the hashing funtion is good, there is very little probability of having collisions between them adn the probability of collision is almost by random, i.e. m/M. Then the number of collisions is, approximately

$$\text{nbr. of collisions} = \sum_{j=0}^{m} j/M \propto \frac{m^2}{M} \tag{163}$$

We consider the following sequence-hasshing functions:

- `Hasher` **(SVID rand48 functions)** If we have some kind of pseudo-random generator in the form $y = \text{rand}(s)$ where s is the seed, then we can do a hashing function with the following seudocode

```
int hash(int *x,int n) {
  int v = 0;
  for (int j=0; j<n; j++) {
    v = rand(f(v,x[j]);
  }
  return v;
}
```

  where `f(v,x)` is some binary functions that combines the values of the current state `v` and the incoming sequence element `x[j]`. The *rand48* hasher is based on the random function based on the SVID library coming with the GNU C library (version 5.3.12 at the moment of writing this).
- `FastHasher` This is based in a simple pseudo-random function of the form

```
int rand(int v,int x) {
  v ^= x;
  int y = (v+x)^2;
  y = y % MAX;
  y = y ^ m;
}
```

```
int hash(int *w,int n) {
  int v = c;
  for (int j=0; j<n; j++) {
    v = rand(v,x);
  }
}
```

where `MAX`=2^{32} and `c = 0x238e1f29`, `m = 0x6b8b4567`.

- `MD5Hasher` This is based on the MD5 routines from RSA. This is an elaborated algorithm that creates a 16 byte hash value from a string of characters. We take as hash value the first 4 bytes from this digest.

18 Synchronized buffer

One difficult task in parallel programming is printing from the slave nodes. In MPI, in general,it is not guaranteed that printing from the nodes is possible and in the MPICH implementation output from the nodes get all mixed and scrambled.

PETSc provides a functions in order to facilitate this task. The user can call `PetscSynchronizedPrintf(...)` as many times as he wants en each nodes. The output is concatenated in each node to a buffer, and then a collective call to `PetscSynchronizedFlush(...)` flushes all the buffers *in order* to the standard output. There is a similar function for files `PetscSynchronizedFPrintf(...)` but it turns out that the flushing of standard output and files is mixed. In addition, even in the case of writing only to standard output, the output is not sorted properly.

The objective of the `SyncBuffer<T>` template class and the `KeyedOutputBuffer` class is to have a synchronized output device that sorts the lines written by the nodes. The idea is that one defines a class (say `KeyedObject`) that must support the following member functions

```
class KeyedObject {
public:
  // Default constructor
  KeyedObject();
  // Copy Ctor
  KeyedObject(const KeyedObject &ko);
  // Dtor.
  ~KeyedObject();
  // Used for sorting the lines
  friend int operator<(const KeyedObject& left,
           const KeyedObject& right);
  //  Call back for the distributed container. Return
  // size of the buffer needed to store this element.
  int size_of_pack() const;
  // Effectively packs the object into the buffer,
  // upgrading the pointer.
  void pack(char *&buff) const;
  // Extracts the object from the buffer, upgading the buffer.
```

```
  void unpack(const char *& buff);
  // Print the object
  void print();
};
```

Then the user can load objects into the buffer, and finally call the `flush()` method to dump the actual content of the buffer to the output. The `flush()` is equivalent to

- sending all objects to the server, deleting them from the original node,
- sorting them according to the `operator<(...)` defined, and
- calling `print(...)` on all of them *on the server.*

The underlying container is a list so that you can manipulate it with the standard list accessors, but the ideal is to push elements with `push_back(KeyedObject obj)`, or pushing a clean object with `push_back()` and then access the elements with `back()`. Note that you must implement the copy constructor for the `KeyedObject` class. Objects with the same key are not overwritten, i.e. if several elements with the same key are loaded on the same or differente processors, then all of them are printed to the output. As the sorting algorithm used is stable, objects with the same key loaded in the same processor, remain in the same order as were entered. Typical usage is as follows

```
#include <src/syncbuff.h>

class KeyedObject {
  // define methods as declared above
};

SYNC_BUFFER_FUNCTIONS(KeyedObject);

int main() {
   SyncBuffer<KeyedObject> sb;

   // Insert objects
   sb.push_back(obj1);
   // ...
   sb.push_back(obj1);

   // flush the buffer.
   sb.flush();
}
```

The macro `SYNC_BUFFER_FUNCTIONS(...)` takes as argument the name of the basic object class and generates a series of wrapper functions. (This should be done in the templates themself but due to current limitations in template specialization it has to be done through macros.)

18.1 A more specialized class

A more simple class derived from `SyncBuffer<...>` has been written. This class is based on `SyncBuffer<...>` using as `KeyedObject` the class `KeyedLine` which has simply an integer and a C-string. Typical usage is

```
#include <src/syncbuff.h>
#include <src/syncbuff2.h>

  KeyedOutputBuffer kbuff;
  AutoString s;

  for ( ... ) {
    s.clear();
    // Load string ‘s’ with ‘cat_sprintf’ functions
    // ....

    // Load in buffer
    kbuff.push(k,s);
  }

  kbuff.flush();
```

Also you can directly use a `printf(...)'` like semantics in the following way

```
#include <src/syncbuff.h>
#include <src/syncbuff2.h>

  KeyedOutputBuffer kbuff;
  int key;

  for ( ... ) {
    // Load internal string with ‘printf’ and ‘cat_printf’ functions
    kbuff.printf("one line %d %d %d %f\n",i,j,k,a);
    kbuff.cat_printf("other line %d %d %d\n",m,n,p);
    // Push in buffer with key ‘q’. ‘push()’ also clears the
    // internal string
    kbuff.push(q)
  }

  kbuff.flush();
```

If you want to dump the buffer on another stream, like a file for instance, then you have to set the `static FILE *KeyedLine::output` field of the `KeyedLine` class. Also, the `static int KeyedLine::print_keys` flag controls whether the key number is printed at the beginning of the line or not. For instance the following code sends output to the `output.dat` file without key numbers.

```
FILE *out = fopen("output.dat","w");
KeyedLine::output = out;
KeyedLine::print_keys = 0;
kbuff.flush();
fclose(out);
```

Also, the member `int KeyedOutputBuffer::sort_by_key` (default 1) controls whether to sort by keys prior to printing, then you can set

```
kbuff.sort_by_key = 0;
```

if you want to disable sorting.

Some notes regarding usage of this class are:

- You can have several `SyncBuffer<..>`'s or `KeyedOutputBuffer`'s at the same time, and you can `flush(...)` them independently.
- **Memory usage:** All items sent to the buffer with `push()` are kept in memory in a temporary buffer. When `flush()` is called all objects are sent to the master, printed and al buffers are cleared. So that you must guarantee space enough in memory for all this operations.
- **Implementation details:** Data is sent from the nodes to the master with point to point MPI operations, which is far more efficient than writing all nodes to a file via NFS. Sorting of the objects by key in the master is done using the `sort()` algorithm of the `list<...>` STL container, which is $O(N \log N)$ operations.

19 Authors

Mario A. Storti (CIMEC) PETSc-FEM kernel, NS and AdvDif modules.

Norberto M. Nigro (CIMEC) PETSc-FEM kernel, NS and AdvDif modules, multi-phase flow.

Rodrigo R. Paz (CIMEC) AdvDif module, AdvDif module, hydrology module, compressible flow, fluid-structure interaction.

Lisandro Dalcín (CIMEC) PETSc-FEM Kernel, Python extension language project scripting extension, linear algebra, preconditioners.

Ezequiel López (CIMEC) Mesh relocation algorithms (mesh-move).

Also many people from the following institutions has contributed directly or indirectly to the generation of this code:

CIMEC *International Center for Computational Methods in Engineering, Santa Fe, Argentina* http://www.cimec.org.ar

INTEC *Instituto de Desarrollo Tecnológico para la Industria Química* http://www.intec.unl.edu.ar

CONICET *Consejo Nacional de Investigaciones Científicas y Técnicas* http://www.conicet.gov.ar

UNL *Universidad Nacional del Litoral* http://www.unl.edu.ar

20 Grants received

The development of this code has been developed under financement from the following grants:

23. *Key:* PICT-2006-VES. *Code:* PICT-1506/2006. *Title:* **Cálculo distribuido en mecánica y multifísica computacional** *Director:* Mag. Victorio Sonzogni. *Financing agency:* FONCyT. *Begin:* 2008. *End:* 2010.
22. *Key:* PICTO-2004. *Code:* PICTO-23295/2004. *Title:* **Integración de procesos del complejo suelo-agua-planta para una mejor planificación hídrica en la cuenca inferior del Río Salado** *Director:* Dr. Leticia Rodríguez. *Financing agency:* FONCyT. *Begin:* 2006. *End:* 2007.
21. *Key:* PIP-2005. *Code:* PIP-5271. *Title:* **Mecánica Computacional en Problemas de Multifísica** *Director:* Dr. M.A. Storti. *Financing agency:* CONICET. *Begin:* 2005. *End:* 2008.
20. *Key:* CAI+D-05. *Code:* CAI+D 2005-10-64. *Title:* **Métodos Numéricos para Resolución de Problemas Multifísica** *Director:* Dr. S. R. Idelsohn. *Financing agency:* Universidad Nacional del Litoral - UN Litoral. *Begin:* 2005. *End:* 2007.
19. *Key:* PROTIC. *Code:* PAV-127, Subproy 4.. *Title:* **PROTIC: Red para la Promoción de las Tecnologías de la Información y las Comunicaciones. Subproyecto 4: Centro Virtual de Computación de Alto Rendimiento** *Director:* Dres. Alejandro Cecatto, Guillermo Marshall. *Financing agency:* ANPCyT - FONCyT. *Begin:* 2005. *End:* 2006.
18. *Key:* LAMBDA. *Code:* PICT 12-14573/2003. *Title:* **LAMBDA: Laboratorio virtual para el Análisis y simulación computacional de problemas Multifísicos Basados en ecuaciones Diferenciales Acopladas** *Director:* Dr. S. Idelsohn. *Financing agency:* ANPCyT - FONCyT. *Begin:* 2005. *End:* 2007.
17. *Key:* PME-CLUSTER. *Code:* PME-209. *Title:* **Cluster del Litoral: Red de laboratorios para la resolución de problemas de la físico-matemática aplicados a la ingeniería** *Director:* Dr. S. Idelsohn. *Financing agency:* ANPCyT - FONCyT. *Begin:* 2004. *End:* 2005.
16. *Key:* CLUSTER-CHILE. *Code:* C-13680/4, Nro 23. *Title:* **Cálculo paralelo en problemas de mecánica computacional a través del uso de una red de computadores personales.** *Director:* Dres. Marcela Cruchaga, Norberto Nigro. *Financing agency:* Programa de Colaboracion Científico-Académica entre Argentina, Brasil y Chile 2000 - 2001 (C-13680/4). Fundación Andes.. *Begin:* 2000. *End:* 2001.
15. *Key:* PIP-PAR. *Code:* PIP 02552/2000. *Title:* **Generación de recursos de cálculo paralelo para mecánica computacional** *Director:* V.E. Sonzogni. *Financing agency:* CONICET. *Begin:* 2000. *End:* 2002.
14. *Key:* CAI+D. *Code:* CAI+D-2000-43. *Title:* **Desarrollo de algoritmos para cálculo paralelo** *Director:* Victorio Sonzogni. *Financing agency:* UNL. *Begin:*

2000. *End:* 2002.

13. *Key:* PROA. *Code:* PICT-6973/99. *Title:* **Desarrollos en Mecánica Computacional utilizando técnicas de PRogramación Avanzada** *Director:* Sergio Idelsohn. *Financing agency:* ANPCyT. *Begin:* 2000. *End:* 2003.
12. *Key:* MELT. *Code:* PID-99/76. *Title:* **MELT: Modelado de Emulsificación de metales en estado Líquido y sus efectos Termomecánicos** *Director:* A. Cardona. *Financing agency:* ANPCyT - FONCyT. *Begin:* 2000. *End:* 2002.
11. *Key:* FLAGS. *Code:* PID-99/74. *Title:* **FLAGS: Simulación numérica en gran escala de la interrelación entre el FLujo de Aguas Superficiales y el FLujo de AGuas Subterráneas** *Director:* S. Idelsohn. *Financing agency:* ANPCyT - FONCyT. *Begin:* 2001. *End:* 2004.
10. *Key:* GERMEN-CFD. *Code:* PIP-0198/98. *Title:* **Germen/CFD: GEneración de Recursos básicos para la aplicación de los Métodos Numéricos en dinámica de fluidos computacional** *Director:* M. Storti. *Financing agency:* CONICET. *Begin:* 1999. *End:* 2001.
9. *Key:* PEI-CFD. *Code:* PEI-231/97. *Title:* **PEI 231 - CONICET. Diseño mecácnico asistido por CFD** *Director:* N. Nigro. *Financing agency:* CONICET. *Begin:* 1998. *End:* 1999.
8. *Key:* PEI-NAVAL. *Code:* PEI-232/97. *Title:* **PEI Nro. 232 - CONICET. Métodos Numéricos en Hidrodinámica Naval y Costera** *Director:* M. Storti. *Financing agency:* CONICET. *Begin:* 1998. *End:* 1999.
7. *Key:* SINUS-PIM-B. *Title:* **Proyecto Alpha de la Comisión Europea. Sinus Pim B: Modelisation et Simulation Numeriques en Ingenierie Mecanique** *Director:* S.R. Idelsohn y V. Ruas (Univ. Paris VI, Laboratoire de modelisation en Mecanique). *Financing agency:* CONICET - CEE. *Begin:* 1997. *End:* 1999.
6. *Key:* CMES. *Title:* **Proyecto Alpha de la Comisión Europea. CMES: Computer Methods in Engineering Science** *Director:* S.R. Idelsohn y G. Beer (Institut für Baustatik, Graz, Austria). *Financing agency:* CONICET - CEE. *Begin:* 1997. *End:* 1999.
5. *Key:* TUCANO. *Title:* **Proyecto Alpha de la Comisión Europea. TUCANO: Transatlantic University / Industry Cooperation** *Director:* S.R. Idelsohn y S. Mac Neill (Univ. de Birmingham). *Financing agency:* CONICET - CEE. *Begin:* 1997. *End:* 1999.
4. *Key:* CONICET-FNRS. *Title:* **Convenio CONICET - Fonds National de la Recherche Scientifique (FNRS) entre el Grupo de Tecnología Mecánica del INTEC y el Laboratoire de Téchniques Aéronautiques et Spatiales, Universidad de Lieja, Bélgica. Investigación en Mecánica Computacional** *Director:* A. Cardona y M. Géradin. *Financing agency:* CONICET - FNRS(Bélgica). *Begin:* 1996. *End:* 1997.
3. *Key:* CAI+D-94. *Code:* CAI+D 94-004-024. *Title:* **Métodos Numéricos en Mecánica de Sólidos y Fluidos** *Director:* S. R. Idelsohn y A. Cardona. *Financing agency:* Universidad Nacional del Litoral - UNLit. *Begin:* 1994. *End:* 1995.
2. *Key:* GERMEN. *Code:* PICT-51. *Title:* **FONCyT - PICT 51 GERMEN: GEneración de Recursos básicos para la aplicación de los MÉtodos Numéricos** *Director:* Dr. Sergio Idelsohn. *Financing agency:* FONCyT. *Begin:* 1998. *End:* 2001.

1. *Key:* COLA. *Code:* PID-026. *Title:* **Simulación Numérica de Procesos de Colada Continua** *Director:* S. R. Idelsohn. *Financing agency:* SECYT-BID. *Begin:* 1996. *End:* 1999.

21 Symbols and Acronyms

21.1 Acronyms

CFD: Computational Fluid Dynamics

DX: IBM Data Explorer

ePerl: embedded Perl

FDM: Finite Difference Method

FEM: Finite Element Method

IISD: Interface Iterative – Sub-domain Direct method

KWM: Kinematic Wave Model (see 6.7)

LES: Large Eddy Simulation

MPI: Message Passing Interface

OOP: Object Oriented Programming

Perl: Practical Extraction and Report Language

PETSc: Portable Extensible Toolkit for Scientific computations.

SUPG: Streamline Upwind/Petrov Galerkin

ANN: Approximate Nearest Neighbor problem. Also refers to the library developped by David Mount and Sunil Arya (`http://www.cs.umd.edu/~mount/ANN`).

22 Symbols

- u, v = Streamwise and normal components of velocity.

References

[1] L. Rodríguez. *Investigation of Stream-Aquifer Interactions Using a Coupled Surface-Water and Ground-Water Flow Model*. PhD thesis, University of Arizona, 1995.

[2] M. Mallet T.J.R. Hughes. A new finite element method for cfd: Iv. a discontinuity-capturing operator for multidimensional advective-diffusive systems. *Comp. Meth. in Applied Mechanics and Engineering*, 58, 1986.